NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

R3662

A PAD ROUTER FOR THE MONTEREY SILICON COMPILER

by

Carlos Francisco Rexach

March 1988

Thesis Advisor: D. E. Kirk

Approved for public release; distribution unlimited.

T239225

ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (If applicable)
7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
. NAME OF FUNDING / SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)
9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO

TITLE (Include Security Classification)

A PAD ROUTER FOR THE MONTEREY SILICON COMPILER

PERSONAL AUTHOR(S)

Rexach, Carlos, F.

a. TYPE OF REPORT	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis		March 1988	184

SUPPLEMENTARY NOTATION

The views expressed in this thesis are those of the author and do not eflect the official policy or position of the Department of Defense or the U.S. Government.

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	VLSI Design, MacPitts, Silicon Compiler, CAD Tools, Pad Router, Pad Placement, Router

ABSTRACT (Continue on reverse if necessary and identify by block number)

A two layer pad router is developed for the Monterey Silicon Compiler. Features include a improved pad placement routine that extracts information from the internal layout to nimize chip area and wiring lengths, and a track allocation algorithm that minimizes the e of polysilicon during net layout. The router's performance was compared to that of the MacPitt's Silicon Compiler with four distinct circuits. The Monterey pad router layouts re 5% to 25% faster, and 10% to 15% smaller than those produced by MacPitts.

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
NAME OF RESPONSIBLE INDIVIDUAL D.E. Kirk	22b TELEPHONE (Include Area Code) (408) 277-9536
	22c OFFICE SYMBOL 62KI

FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-243

Approved for public release; distribution is unlimited

A Pad Router for the Monterey Silicon Compiler

by

Carlos Francisco Rexach
Lieutenant, United States Navy
B. S., University of Puerto Rico, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1988

ABSTRACT

A two layer pad router is developed for the Monterey Silicon Compiler. Features include an improved pad placement routine that extracts information from the internal layout to minimize chip area and wiring lengths, and a track allocation algorithm that minimizes the use of polysilicon during net layout. The router's performance was compared to that of the MacPitts silicon compiler with four distinct circuits. The Monterey pad router layouts were 5% to 25% faster, and 10% to 15% smaller than those produced by MacPitts.

Thesis
R3662
c.1

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
B. SCOPE OF THESIS INVESTIGATION 2
C. THESIS ORGANIZATION 2
II. ROUTING TECHNIQUES 4
A. GENERAL ROUTING PROBLEM 4
B. ROUTERS STUDIED 9
1. Lee Router 9
2. Global Router 15
3. Channel Router 23
4. River Router 29
5. Moat Router 32
C. SUMMARY 34
III. ROUTING IN MACPITTS 35
A. ROUTING IN DATA-PATH 35
B. ROUTING BETWEEN DATA-PATH AND CONTROLLER 36
C. ROUTING TO PADS 43
1. Pad Placement 44
2. Net Extraction 46
3. Net Layout 49
D. SUMMARY 54
IV. ROUTING IN THE MONTEREY SILICON COMPILER 55
A. DATA-PATH 57
B. PAD ROUTING 58

1. Pad Placement 60
2. Pad Layout 69
3. Net Extraction 73
4. Net Layout 75
C. SUMMARY 82
V. RESULTS 83
A. MONTEREY SILICON COMPILER ENVIRONMENT 83
B. RESULTS 83
1. Area 93
2. Wire Length 93
3. Vias 98
VI. CONCLUSIONS 99
A. SUMMARY 99
B. RECOMMENDATIONS 100
APPENDIX A: MACPITTS' FUNCTIONS 101
APPENDIX B: MONTEREY FUNCTIONS 133
APPENDIX C: SOURCE CODE FOR TEST CIRCUITS 165
A. MEMORY 165
B. TEST 165
C. MULTIP4 166166
D. TAXI 167
LIST OF REFERENCES 168
BIBLIOGRAPHY 171
INITIAL DISTRIBUTION LIST 172

LIST OF FIGURES

1. Mead-Conway width and spacing rules 8

2. Lee router example initial circuit configuration 12

3. Cell configuration after Lee algorithm with a minimal- crossing parameter is applied 14

4. Cell Configuration after Lee algorithm with a minimum- wire-length parameter is applied 15

5. Partitioning a chip into global wiring cells 17

6. Grid with original boundary costs 18

7. Cell values after step 5 19

8. Cell values after step 6 20

9. Cell values at end of forward propagation 21

10. The backtrace route 22

11. The left-edge channel router 24

12. A channel with vertical constraints 25

13. Making the vertical constraint graph for the channel of Figure 8 27

14. Vertical constraint loop 28

15. Channel routed by the greedy algorithm 29

16. A channel unroutable by the greedy algorithm 30

17. The river routing problem 31

18. Concentric tracks and radial column geometry of moat routing region 33

19. Moat routing direction ambiguity 33

20. MacPitts data-path design 36

21. Typical MacPitts controller design 38

22. Wing layout 40

23. Solution to river routing problem 41

24. Routing pads in MacPitts 45

25. MacPitts floor plan 51

26. Routing moat corners in MacPitts ..... 53.

27. Typical MacPitts circuit design 56

28. Opening data-path on the left and right sides 59

29. MacPitts' pad ring 61

30. MacPitts' power and ground frame 63

31. Circuit with pads on four sides 65

32. A circuit with pads on two sides 68

33. Pins-layout 70

34. Contents of layout-pad20b-input-pad 72

35. Pad routing area 76

36. Sample routing problem 78

37. Routing the corners 81

38. MEMORY design by MacPitts 86

39. MEMORY design by Monterey 87

40. TEST design by MacPitts 88

41. TEST design by Monterey 89

42. MULTIP4 design by Macpitts 90

43. MULTIP4 design by Monterey 90

44. TAXI design by MacPitts 91

45. TAXI design by Monterey 92

46. Sample MOS circuit 94

47. Bounds for the step response of circuit in Figure 44 for various lengths of polysilicon interconnect: (a) $L = 0$, (b) $L = 100\mu$ 95

48. Bounds for the step response of circuit in Figure 44 for various lengths of polysilicon interconnect: (a) $L = 1mm$, (b) $L = 1cm$ 96

LIST OF TABLES

1. Statistics for MacPitts and Monterey chip designs 85
2. Guidelines for ignoring RC wire delays 94

I. INTRODUCTION

A. BACKGROUND

Weste and Eshragian [Ref. 1] describe silicon compilers as "an automatic translation tool that converts a behavioral description into a mask level description." Silicon compilers provide a powerful tool that allows the designer to explore performance tradeoffs associated with changes to VLSI designs. This is a capability that less automated design environments do not allow.

The Monterey silicon compiler (MSC) evolved from an ongoing effort to convert the NMOS based MacPitts silicon compiler into a Scalable Complementary Metal Oxide Silicon (SCMOS) silicon compiler. MacPitts was developed by Siskind. Southard and Crouch [Ref. 2] at the Massachusetts Institute of Technology Lincoln Laboratories in 1981 – 1982. Like its predecessor, the MSC is a fixed floor plan silicon compiler suited to handle concurrent parallel data-path architectures common in many signal processing applications.

MacPitts has been studied at the Naval Postgraduate School since 1984. Early thesis work dealt with its installation and documentation. Very little documentation existed on MacPitts at that time. MacPitts was installed on the VAX-11/780 by D. Carlson in 1984 [Ref. 3]. In 1985 A. Froede [Ref. 4] discussed MacPitts' internal structure, and R. Larrabee [Ref 5], demonstrated the relationship between the source program and the final chip layout. In 1986, M. A. Malagon-Fajar [Ref. 6] completed a valuable study on the relationship between the compiler and its layout language, L5. In the same year, E. Weist [Ref. 7] developed a flowchart based input interface for MacPitts. In 1987, A. Mullarky [Ref. 8] designed the first SCMOS cells, and E. Malagon [Ref. 9] described the structure of the data-path and inserted

the first SCMOS organelles. That same year, J. Baumstarck [Ref. 10] designed and inserted additional SCMOS organelles.

B. SCOPE OF THESIS INVESTIGATION

MacPitts' pad placement and pad routing algorithms are tremendously inefficient in both area and speed performance. The order in which pads appear on the chip is specified by the user in the source file. MacPitts distributes the pads as evenly as possible along the top, right and bottom sides. No effort to optimize any of a number of possible parameters is attempted.

A second significant problem with MacPitts' designs is the requirement for all nets connecting to pads to enter the circuit through the left side. The extremely long routing paths that result, impact adversely on the chip's speed.

This investigation has two goals. First, to continue the study on the structure and methods of MacPitts, and, second, to develop, implement and test algorithms that will do a better job of routing pads than in MacPitts. This thesis introduces and documents a set of LISP functions that result in more efficient pad placement and routing. This is accomplished by first opening up the internal circuit to the outside on both the left and right sides. Second, area and wire length optimization criteria are introduced into the pad-placement routines. Finally, net layout algorithms were modified to minimize the use of inferior routing layers, such as polysilicon.

C. THESIS ORGANIZATION

Chapter II discusses various routing methods. The routers discussed were selected from the many available because of their applicability to specific routing issues within MacPitts, or because of their fundamental value. Chapter III describes the pad and river router and pad placement used by MacPitts, and Chapter IV describes a new pad placement and pad router. Chapter V gives the comparative analysis

results between the new and old pad placement and pad routing techniques. Conclusions drawn from the results of Chapter V, and suggestions for future research are offered in Chapter VI. Appendix A contains the LISP code of the functions in MacPitts involved with pad routing and placement. Appendix B contains the LISP functions that implement the new pad placement and routing process. Appendix C includes the source files of all the circuits tested.

II. ROUTING TECHNIQUES

Much work has been done on the LSI and VLSI interconnection problem. The realization that all-purpose, optimal routers are an impractical approach to the interconnection problem has led to a search for different approaches. The solution has been to develop routers specialized to interconnect specific geometries. These routers produce near optimal solutions with reasonable resource requirements by exploiting circuit characteristics. This chapter discusses the nature of the VLSI and LSI interconnection problem and surveys current routing techniques.

A. GENERAL ROUTING PROBLEM

The input for an instance of an IC layout problem consists of a set of cells and a set of signal-net definitions. A cell can be thought of as a rectangular box with pins on its boundary. Pins specify the location on the cell perimeter where electrical connections are made, and a signal-net identifies a set of pins to be interconnected. As a general rule, routing paths are not allowed to cross over cells. Because of its complexity, the custom IC layout problem is divided into a placement phase and a routing phase. In the placement phase the objective is to find a cell arrangement that leads to an "optimum" circuit layout. It must minimize layout area, yet allow sufficient space between cells for efficient routing. Several interesting techniques have been developed. With names like *synthetic annealing* and *genetic evolution*, these techniques emulate natural processes and avoid entrapment in local minima by introducing a degree of randomness into the optimization procedure. This study assumes that an acceptable cell placement has already occurred.

Once cell locations are established the routing process begins. The routing problem is defined by a set of cells and a set of signal nets. A solution to the routing

problem is obtained once all pins specified by the signal-nets are interconnected. The interconnection process must be completed within the context of technology dependent design rules.

The previous statement of the routing problem disguises its complexity. Finding an optimal solution to routing even modest-sized circuits is an extremely difficult task. In fact, routing problems belong to a large class of NP-complete (nondeterministic polynomial time complete) problems.[1] No method for an exact solution with a computing effort bounded by a power of N, where N is the number of interconnections, is known.

Many factors contribute to the difficulty in finding optimal routing solutions. The most significant are related to the physical and electrical properties of the materials used in the circuit and to limitations in the fabrication process. The effect these issues have in the design process is often included in the design environment adopted by the designer. This environment is embodied by a small set of design rules.

The properties of the various layers used in IC manufacture and their interaction determine which layers are suitable for routing and how they are to be used. An ideal routing layer has negligible propagation delays and is electrically insulated from all other layers. Of course, no single layer measures up to this ideal. However, as the following description shows, some layers come closer to achieving this ideal than others.

An n-channel metal-oxide-semiconductor (nMOS) fabrication technology uses metal, polysilicon and diffusion layers. Because the metal layer is insulated from all others, it can cross either diffusion or polysilicon with no significant functional

[1] NP-complete problems do not yield optimal solutions to any efficient algorithm. The name refers to the empirical observation that no solution bounded in time by a polynomial in S, where S is the number of steps required to solve one instance of the problem, has been found.

effect. In fact, to connect metal to other layers requires a poly-metal or a diffusion-metal cut. Cuts are little more than holes through the insulating layer that permit electrical connections between dissimilar layers. Unlike metal, whenever polysilicon completely crosses diffusion, an nMOS field effect transistor is created.

Various fabrication and operation mechanisms contribute to alter the final product from that conceived by the designer. These include: mask misalignment, variations in the photoresist edges due to variations in exposure, undercutting of the oxide beneath photoresist corners, overetching, spreading of diffusion and implantation under gates or near the source drain end, and tolerance of the field-oxide windows. Finally, a feature's size can change during operation due to metal migration.

Rather than struggle with a long list of complex process and fabrication dependent parameters the designer works in an environment consisting of a few conservative *design rules.* Design rules can be thought of as allowable mask layer geometries that permit design variations while guaranteeing correct circuit behavior. The purpose of design rules is to guarantee that, under the cumulative contributions of those factors mentioned in the preceding paragraph, the circuit operates correctly. There are many different design rule sets in use today. It is up to the user to select those rules that best meet the requirements established by the technology, fabrication process and intended market.

In 1979 Mead and Conway [Ref. 11] formulated a set of design rules for the nMOS process that have become a standard in academia. They rely on the length parameter λ to determine a minimum feature size. The quantity λ corresponds to the maximum deviation of a mask on the wafer from its intended position. As a result, the maximum deviation of two features on different masks on a wafer is 2λ. If the crossing of these two features is catastrophic for the design, they must be separated by at least 2λ in the design drawing. If not, it is assumed that the

maximum movement of an edge is 0.5λ and the edges must be separated by at least 1λ. These two rules form the basis for most of the Mead-Conway design rules. Stricter exceptions are needed to contend with metal width and spacing, and diffusion spacing. Mead-Conway design rules specify the minimum conducting path width and the minimum distance between any layer combination. Simplified Mead-Conway rules are depicted in Figure 1.

The decision to use Mead-Conway rules was based on three factors: they are relatively simple to use; they are widely accepted by the academic community; and they are the design rules which the MAGIC [Ref. 12] graphics editor design rule checker uses. It should be noted that circuit performance degradation, as well as an increase in chip area could make the λ rules unsuitable for commercial use. The original MacPitts designs did not adhere to Mead-Conway design rules. Lt. J. Harmon edited the organelles.l and the control.l files at the Naval Postgraduate School to remove all non-conforming structures.

The routing problem is further complicated by a wide choice of optimization parameters. M. F. Kelly [Ref. 13] conducted a study on comparative router performance. In the study he lists 22 problem specifications (routable area, number of net-lists, etc.) and 22 performance description characteristics (total wire length, routing area, etc.). Since there exists no practical algorithmic router, there is no routing technique available that will route every circuit optimally. A router that performs well for a specific circuit geometry and a given set of parameters may yield unfavorable results if either the circuit architecture or optimization parameters change. Various parameters can be used to evaluate router performance. Candidates include:

1. Degree of automation

2. Total wire length

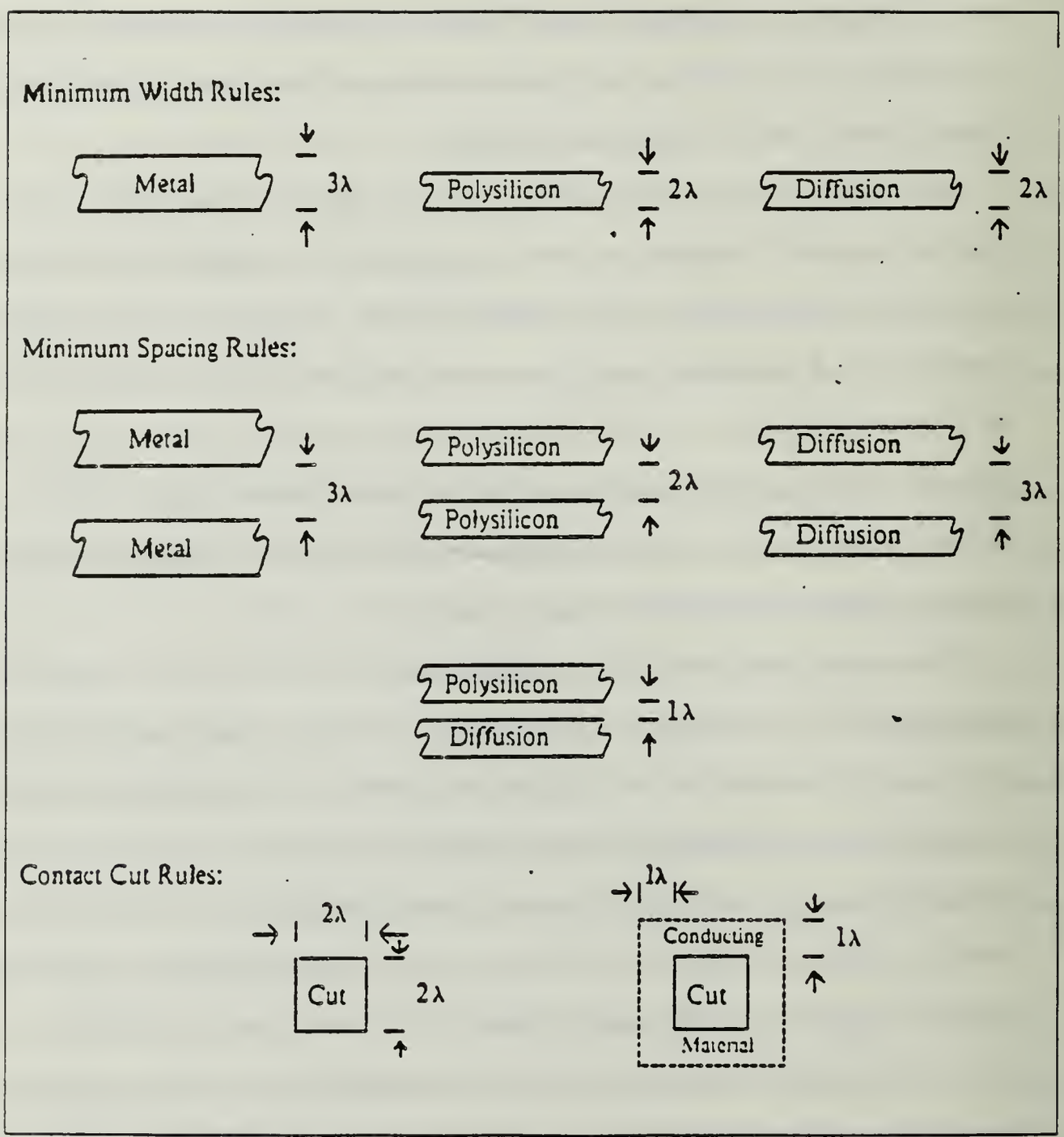

Figure 1: Mead-Conway width and spacing rules

3. Total resistance and capacitance

4. Number of cuts

5. Total routing area

6. Wire density or congestion

7. Number of bends

8. Computer resources required

B. ROUTERS STUDIED

Many routers have been developed to solve circuit interconnection problems. By one set of criteria they are divided into two categories: *algorithmic* or *heuristic*. The algorithmic suite is represented by Lee's [Ref. 14] *expanding wave router* and its variants. Lee's router is algorithmic in that it guarantees to find a path for a given net provided a path exists. All other routers are based on some heuristic.

A different set of criteria partitions the router suite into either *detailed* or *global* routers. Global routers identify those routing areas that a net must traverse to make contact with all its terminals. Detailed or exact embedding routers complete the routing job by actually laying out the conducting paths.

A problem encountered in selecting those routers to be included in this study when comparing routing strategies is the large numbers of routers currently available. Since it is virtually impossible to investigate all available routers, a selection was made based on their fundamental value or their applicability to routing issues in MacPitts. The routers to be discussed are the *Lee router, global router, channel routers, river router* and the *moat router.*

1. Lee Router [Ref. 14]

Lee's router resulted from an endeavor to "... find procedures that will enable a computer to solve efficiently path connection problems inherent in logical drawing,

wiring diagramming and optimal route finding." It is flexible enough to permit the user to choose among one or more optimization parameters: wiring length, congestion, layer crossover, etc. All versions route one net at a time, starting out with a pin of the net and progressively assigning distance values to the cells surrounding it, as in an expanding wave. This continues until the destination pin is reached. The optimum path is marked by backtracking towards the source pin by way of those cells with the minimum distance values. Variations between versions of Lee's algorithm are primarily the result of different attempts to speed up the routing process.

All Lee routers suffer from three significant flaws. First, since each net is routed independently, early routing decisions can significantly degrade or completely block the routing of subsequent nets. Second, Lee routers require vast amounts of storage since each cell must be able to store both a distance value and information about a backtrack direction. Finally, Lee routers are slow since the wavefront expands in all directions irrespective of the target's direction. Generally, the algorithm has a time complexity in the order of $O(n^2)$, where n is the minimum distance between source and target.

Lee translates the layout problem into a C-space defined by the quintuple (C,S,N,Γ,M). In the model:

1. C is the set of cells, $C = c^1, c^2, ..., c^n$ defined by a grid superimposed on the routable area.
2. N is the 1-neighborhood function, $N(c^i) = c^i_1, c^i_2, ..., c^i_n$. $N(c^i)$ defines a subset of C whose elements are physically adjacent to c^i.
3. S is a finite set of symbols called the alphabet of C.
4. Γ is a map of C to $C \times S$. For every $c^i \in C$, $\Gamma(c^i) = (c^i, s^j)$, $s^j \in S$. In other words, for every cell $c^i \in C$ there is mapped to it a symbol $s^j \in S$.
5. Let c^i and c^j be two distinct cells in C. By a path $p(c^i, c^j)$ is meant the chain of all cells $c^0 = c^i, c^1, c^2, ..., c^m = c^j$ such that $c^{i+1} \in N(c^i)$ for $(i = 0, 1, ..., m)$. By $\pi(c^i, c^j)$ is meant the set of all paths $p(c^i, c^j)$ between cells c^i and c^j.

6. M is the *admission map* with domain $\pi(c^i, c^j)$ and range $0, 1$. Any path $p(c^i, c^j)$ is said to be admissible if $M(p(c^i, c^j)) = 1$. Otherwise the path is said to be inadmissible. The set of all admissible paths is denoted by $\pi^*(c^i, c^j)$.

Given a set of admissible paths, Lee's algorithm finds the minimal path with respect to a vector of r functions $F = (f_1, f_2, ..., f_r)$. A path $p^1(c^i, c^j)$ is said to be minimal with respect to f_1 if

$$f_1(p^1(c^i, c^j)) \leq f_1(p(c^i, c^j))$$

for all $p(c^i, c^j) \in \pi^*(c^i, c^j)$. A path is minimal with respect to (f_1, f_2) if $p^{12}(c^i, c^j) \in P^1(c^i, c^j)$, where $P^1(c^i, c^j)$ is the set of all paths in $\pi^*(c^i, c^j)$ that are minimal with respect to f_1. In other words, p^{12} is minimal with respect to (f_1, f_2) if along all paths minimal in f_1, it is also minimal in f_2. In a similar fashion, a path $p^{123...r}(c^i, c^j)$ can be found that is minimal with respect to $(f_1, f_2, ..., f_r)$. The vector F represents the parameter or parameters which are to be minimized. Priority between parameters is determined by position in F. The first parameter to be evaluated has the highest priority, the second is next, and so forth. This convenient ordering results because the minimal subset of $\pi^*(c^i, c^j)$ is first selected with respect to f_1. All admissible paths resulting from this selection become the domain from which minimal paths for subsequent parameters are selected. This sequence of dependency is repeated for each function.

Given two cells $(c^i, c^j) \in C$, Lee's algorithm will find those admissible paths which are minimal with respect to F. The algorithm simply starts at one end, say c^i and marks all $N(c^i)$ with a symbol of S. It then looks at all $N(N(c^i))$ and marks all that are admissible (not a member of $N(c^i)$). It continues this procedure until it reaches the target cell or until all cells in C are marked. In this case, no path exists. Finally, from all the possible solutions the optimal one is selected.

The following example illustrates how Lee's algorithm works. In this example the goal is to find the path between terminals of a net that minimize the number

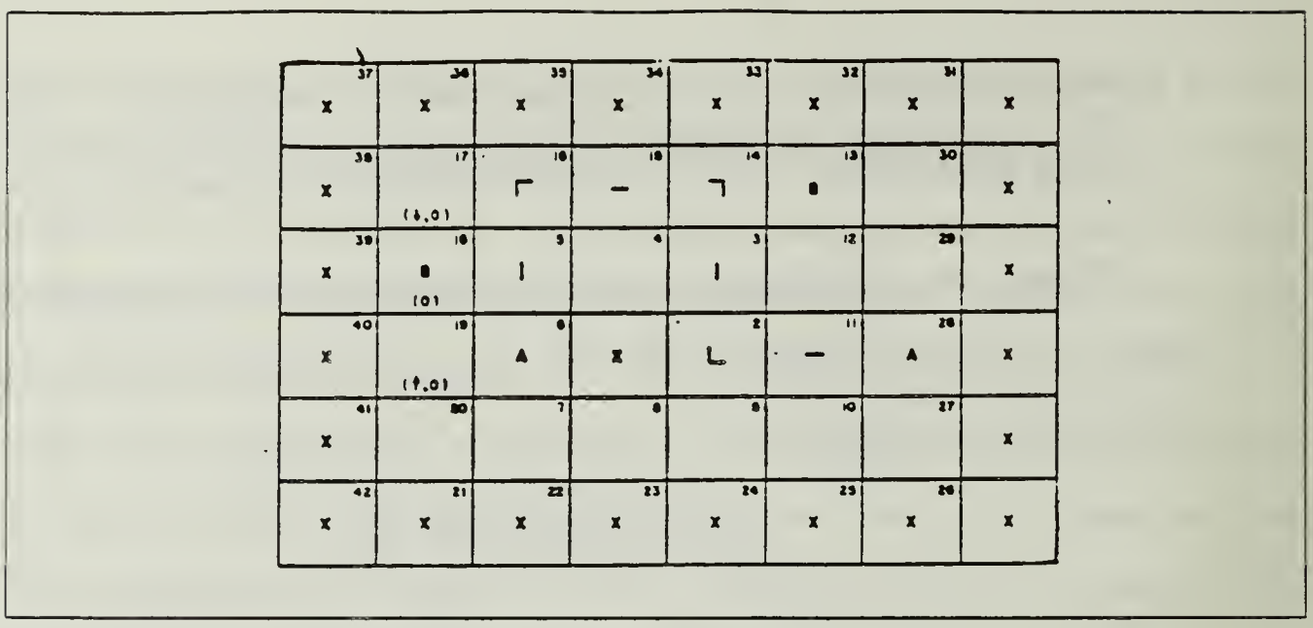

Figure 2: Initial circuit configuration

of other net paths crossed. Lee introduced it as a minimal-crossing problem in his original paper. Actually, it is a two function problem because it also prevents routing on the edges (squares marked with x). In the example, net A has already been routed through squares 6, 5, 16, 15, 14, 3, 2, 11 and 28 (see Figure 2).

The 1-neighborhood function N and coordinate functions d_1, d_2, d_3, and d_4 are defined as follows. Given a cell c^i, $d_1(c^i)$ is the cell directly above c^i, $d_2(c^i)$ is the cell to the right of c^i, $d_3(c^i)$ is the cell below c^i and $d_4(c^i)$ is the cell to the left of c^i.

The vector F is constructed by considering the parameters to be optimized and the properties of the fabrication materials. For example, in a two routing layer technology such as nMOS, the routing process can be simplified by assigning one routing layer to horizontal wires and the other to vertical layers. If a wire has to change direction a via is used to connect the two dissimilar layers. For such a minimal crossing problem the vector F consists of one function f which is described as follows:

1. $f(p(c^*, c^*)) = 0$. The cell mass of a cell routed to itself is 0.

2. If $s(c^i)$ = blank, then

$$f(p(c^*, c^i)) = \begin{cases} \min_{\{f(p(c^*,c^j)),|c^j \in N(c^i)\}} \text{ over all } c^j \\ \text{for which } f(p(c^*,c^j)) \text{ has been defined;} \\ \text{undefined otherwise} \end{cases}$$

The cell mass of a blank cell equals the minimum among the cell masses of its 1-neighbors.

3. If $s(c^i) = —$, then

$$f(p(c^*, c^i)) = \begin{cases} (\min_{\{f(p(c^*,d_1(c^i))),f(p(c^*,d_3(c^i)))\}})+1 \\ \text{if either one of the values of } f \text{ is defined;} \\ \text{undefined otherwise} \end{cases}$$

The cell mass of a cell with a vertical wire is the minimum among the cell masses of its 1-neighbors plus one.

4. If $s(c^i) = |$, then

$$f(p(c^*, c^i)) = \begin{cases} (\min_{\{f(p(c^*,d_2(c^i))),f(p(c^*,d_4(c^i)))\}})+1 \\ \text{if either one of the values of } f \text{ is defined:} \\ \text{undefined otherwise} \end{cases}$$

The cell mass of a cell with a horizontal wire is the minimum among the cell masses of its 1-neighbors plus one.

5. If $s(c^i) = \lfloor, \rfloor, \lceil$ or $\rceil$, then $f(p(c^*, c^i)) = \infty$ The cell mass of a cell with a via is large enough to prevent routing through it.

6. If $s(c^i)$ = x, then $f(p(c^*, c^i)) = \infty$ The cell mass of edge cells is large enough to prevent routing through it.

The following terms are used throughout the discussion of the minimal crossing example:

1. A cell list L consists of names of cells which have been looked at and still have admissible 1-neighbors that have not been assigned a chain coordinate.

2. Cell mass is the sum of f as it follows the path from the source cell to the current cell being examined.

3. Chain coordinates are assigned to each cell that has been investigated. They specify both the direction to the previous cell and the cell mass.

A solution to the routing of net B in Figure 2 is shown in Figure 3. The initial L list starts out with cell 18, the source cell. Admissible 1-neighbors include

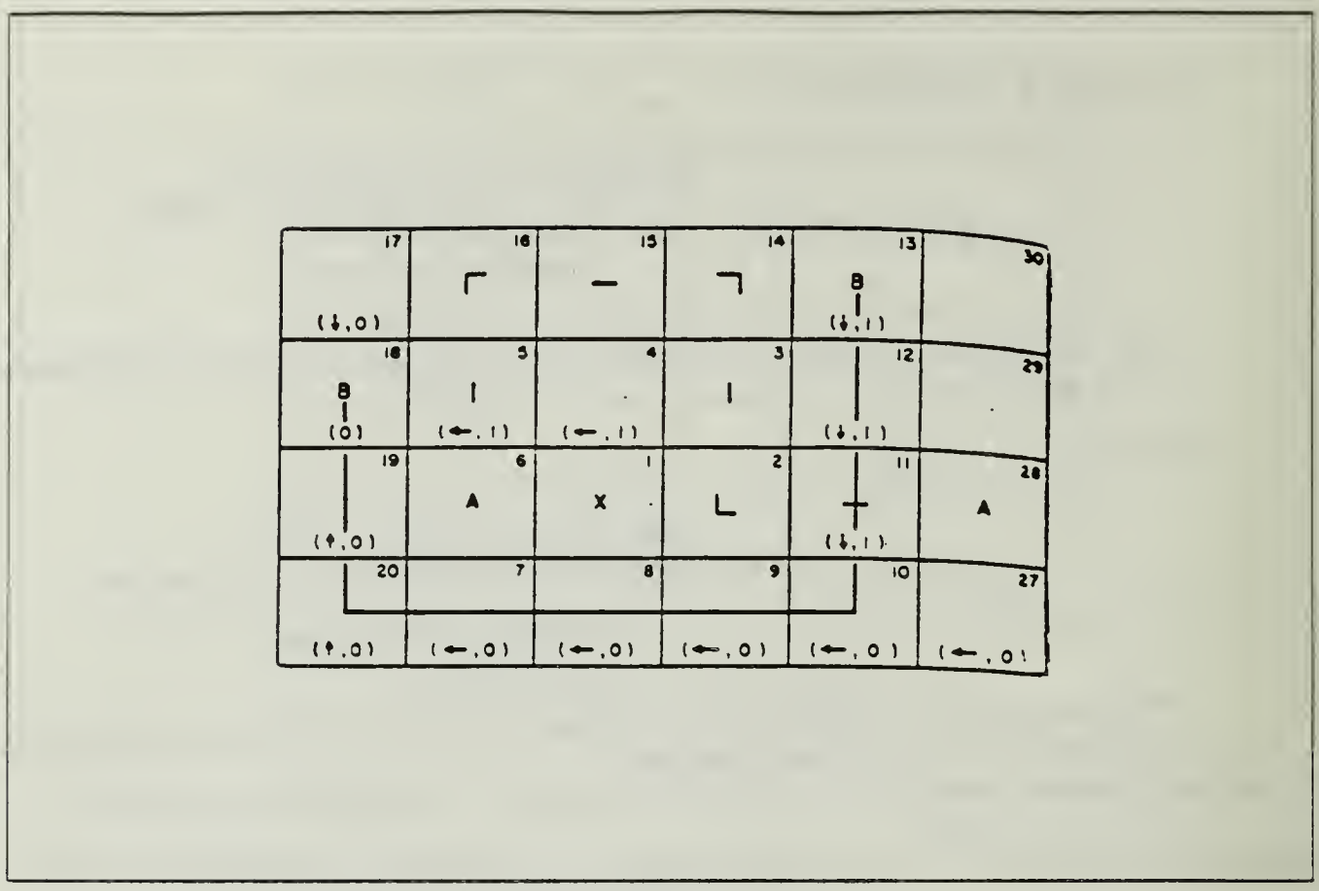

Figure 3: Cell configuration after Lee algorithm with a minimal-crossing parameter is applied

[5, 17, 19]. Notice that cell 39 is not an admissible neighbor of 18 because it sits on the edge. Based on the definition of f, cell 5 is assigned a cell mass of 1 because it already has a vertical wire. Cells 17 and 19 have cell masses of 0 because they are blank cells. Since their cell masses are minimum, cells 17 and 19 are appended into L. Furthermore, chain coordinates ($\downarrow$, 0) and ($\uparrow$, 0) are assigned to 17 and 19, respectively. The 0 in the chain coordinate refers to the cell mass. The arrow points the path back to the source cell. L now consists of cells 17, 18 and 19.

The process is repeated. Now the admissible 1-neighbors to list L include 5 and 20. All other cells are inadmissible because they are either edges (cells 36, 37, 38, 39 and 40), have vias (cell 16) or have terminals (cell 6). The cell masses assigned are 1 and 0 respectively. This time, cell 20 is added to L and is assigned a coordinate chain of ($\uparrow$, 0). Also, 17 and 19 are removed from L since all of their possible 1-neighbors are either inadmissible or have alredy been assigned a chain

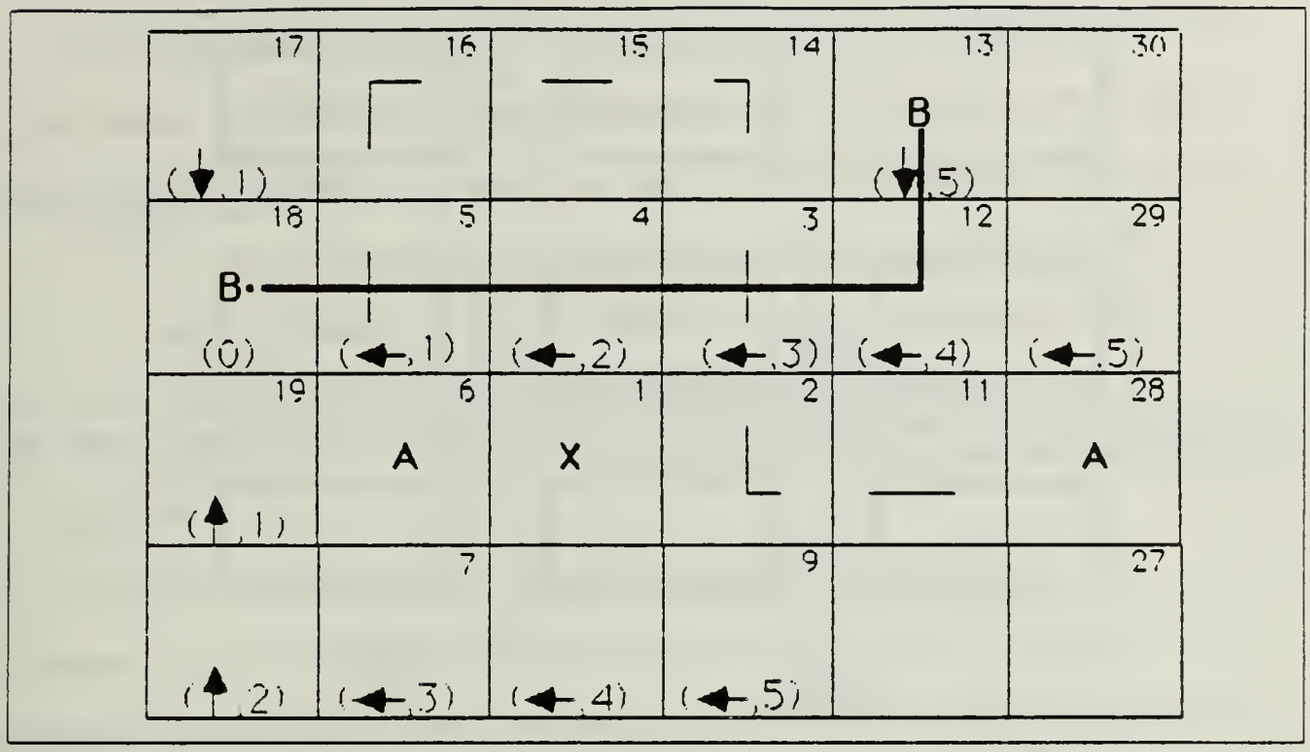

Figure 4: Cell configuration after Lee algorithm with a minimum-wire-length parameter is applied

coordinate. Repeating this process leads the pointer to cell 27. L now includes cells 18, 10 and 27. Admissible 1- neighbors are cells 5 and 11 with cell masses of [1, 1], respectively. L now consists of cells 5 and 11. The algorithm has determined that it is not possible to route net B without crossings. Chain coordinates of ($\leftarrow$, 1) and ($\uparrow$, 1) are assigned to 5 and 11, respectively. The process continues until the path with minimum cell mass reaches the target cell as shown in Figure 3. Notice that this path has one crossing at cell 11.

Figure 4 shows the same problem routed with respect to a minimum-wire-length parameter. In this instance, different parameters yield different results for the same circuit. It is precisely this flexibility that has made Lee's algorithm one of the most popular Printed Circuit Board routers.

2. Global Router

Global routers assign nets to routing regions but leave the actual track layout to exact embedding routers such as the Lee or channel router. They are effective in reducing seemingly intractable problems, into a series of less complex ones. This

divide and conquer approach can reduce computation times drastically. Global routers must provide certain essential information for the exact embedding router to complete the routing process. At a minimum this information must include:

1. The set of routing regions traversed by each signal net.
2. The set of edges crossed by a signal-net in each routing region.
3. A set of strands specifying which crossings must be connected within a region. A net may have more than one strand within a region. Such cases result in tree-like interconnection structures.

The global router algorithm is similar to the Lee router algorithm. Like Lee's algorithm, the router is concerned with one path at a time. It begins at the source and expands outward, evaluating a cost at each routing area. The algorithms differ in that global routers evaluate all routing areas in the circuit (using Lee router terminology, every routing region is assigned a cell mass for every net-list). Once cell values are assigned, the optimal path is determined by backtracking from destination to source as in the Lee router. Before starting a more detailed description of the algorithm, some definitions are in order:

1. Routing regions must be defined by the global router or by some prior algorithm. A common practice is to extend the horizontal and vertical cell boundaries until obstacles (other cell boundaries) are reached. The rectangular areas that form from the intersections of the various edge extensions are the routing regions (see Figure 5).
2. Channel capacity refers to the number of crossings that a routing region can support.
3. A fence-list consisting of all nets having pins within a routing region is constructed to ensure that crossing space is available for these cases regardless of when that signal-net is routed.
4. Edge values represent the cost assigned a net-list for traversing any routing region edge. This is a dynamic value. As more nets are allowed to cross a channel, the channel edge values increase. When the channel is saturated, an arbitrarily large number representing infinity is given to that region's edges to prohibit any other nets from using that channel.

The global router initiates its path search at the signal-net source and expands in all directions (vertical and horizontal) until all routing regions have been tested.

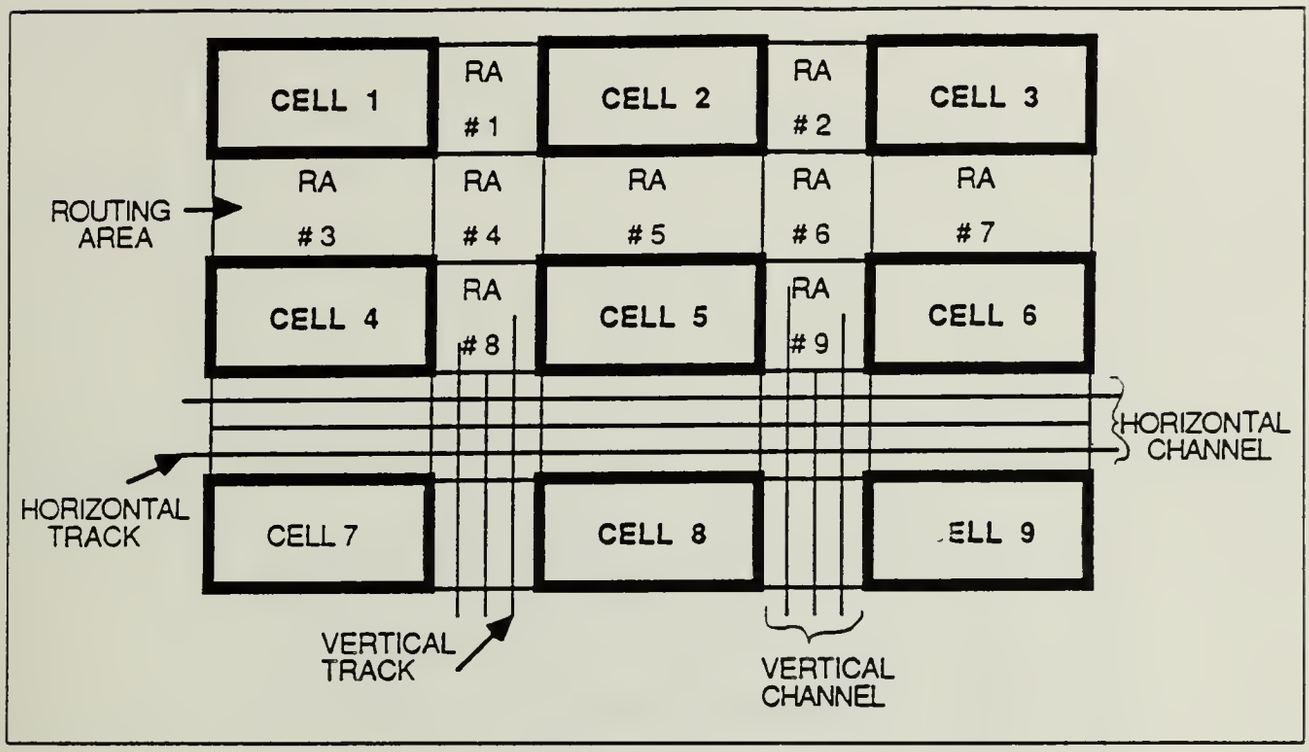

Figure 5: Partitioning a chip into global wiring cells.

At each region, a value representing the total routing cost from the source to the current region, is calculated. A cost value is computed by adding the cost of crossing into the new area to the value calculated at the previous area. If this value is less than the current value for that cell, it becomes the new value. Otherwise, the old value remains in force. Once all regions are assigned a cost, the optimum path is determined by backtracking along the minimum cost routing areas until the source is reached. As with the Lee router, different optimization parameters can be used to meet specific design goals.

Figures 6 – 10 describe events that lead to the selection of a path for net A. The initial edge values in Figure 6 are the result of nets that have already been routed. Scores at the fifth and sixth step and at the end of the forward propagation are shown in Figure 7 – 9. The number inside each square is the minimum sum of all the edges crossed to get to that square from the source cell. Consider, for example, square (4,2), i.e., the cell in the 4th column and 2nd row. The cost at square (4, 2) at step six is the minimum of the cost of moving to (4,2) from square (3,3), which

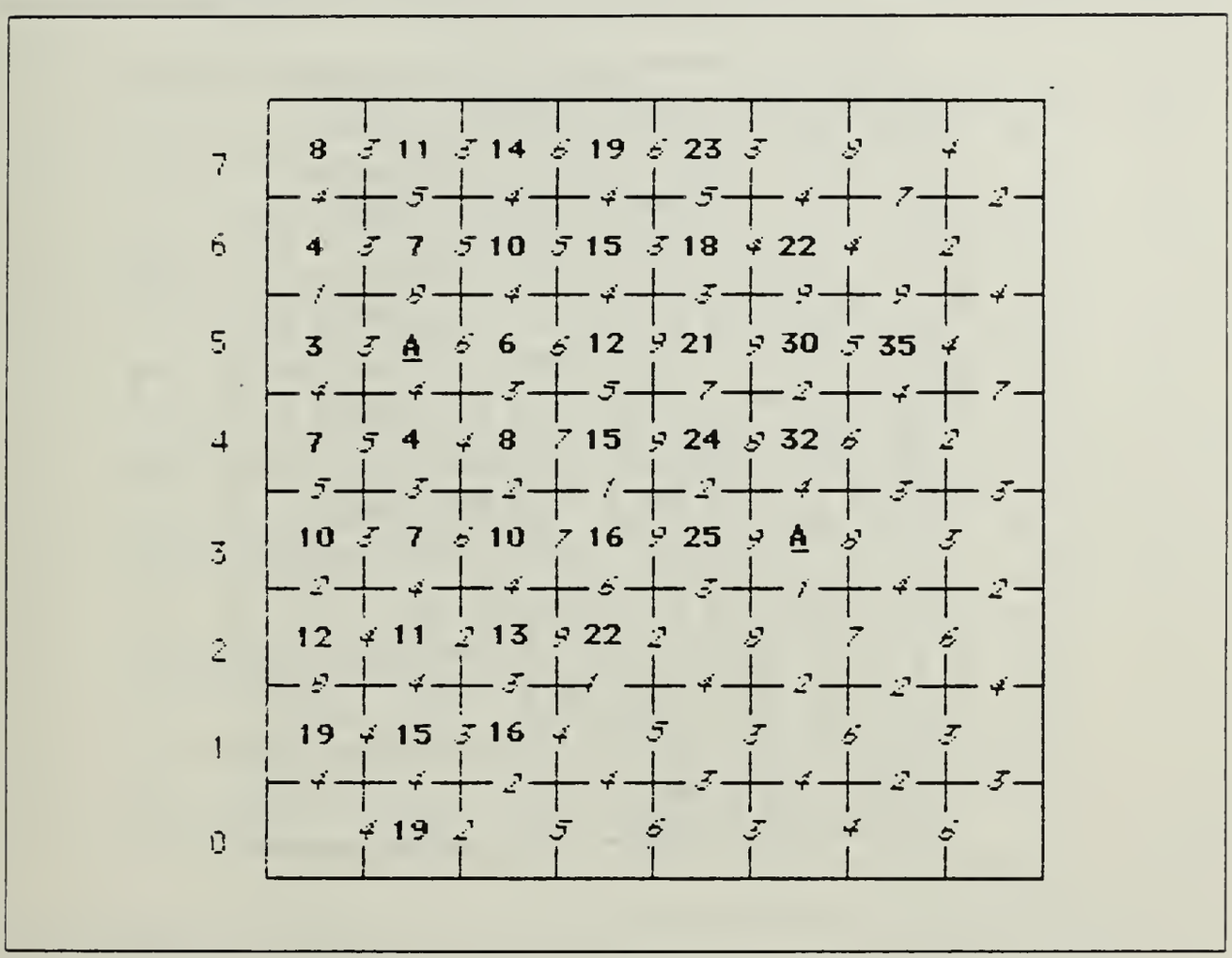

Figure 7: Cell values after step 5.

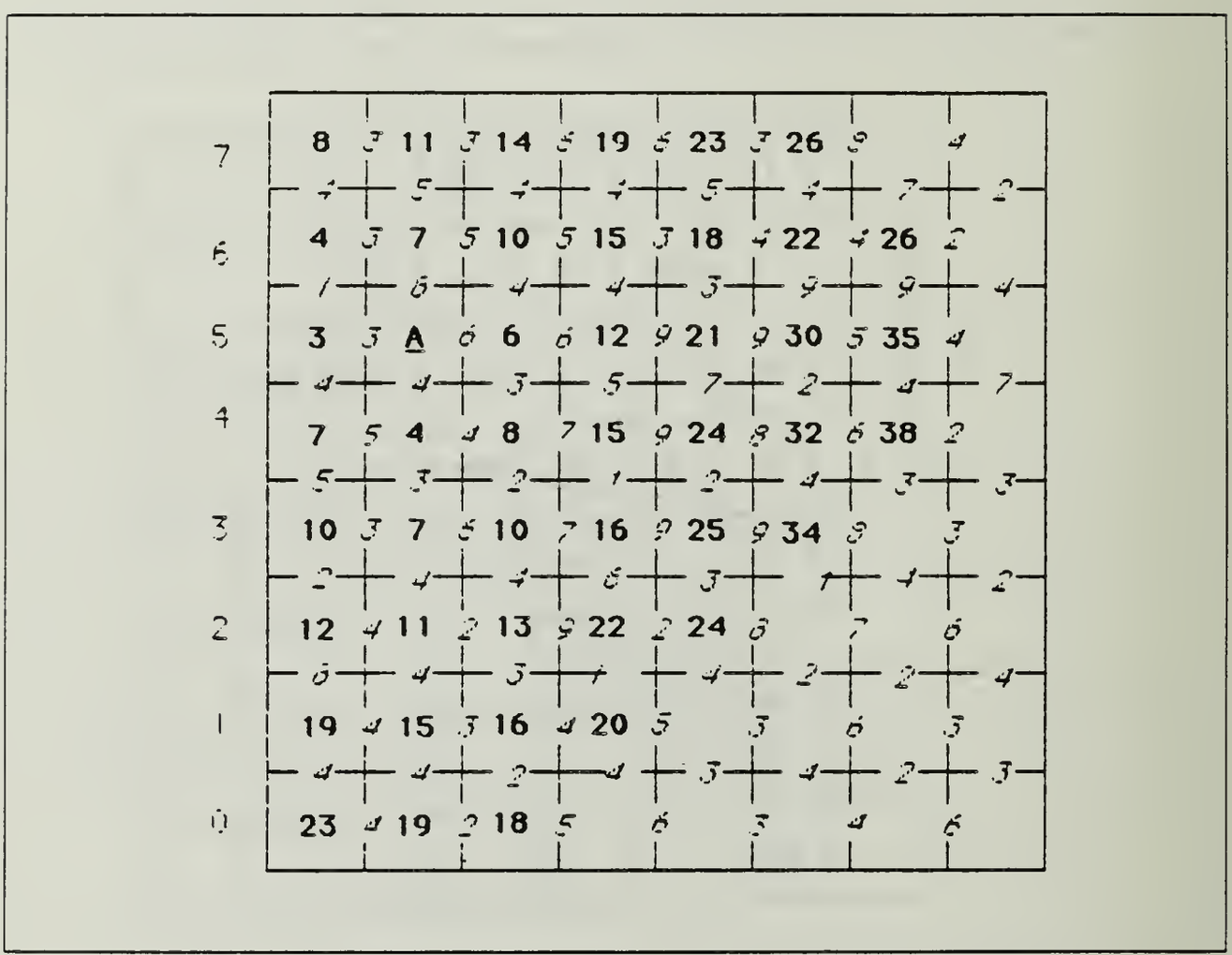

Figure 8: Cell values after step 6.

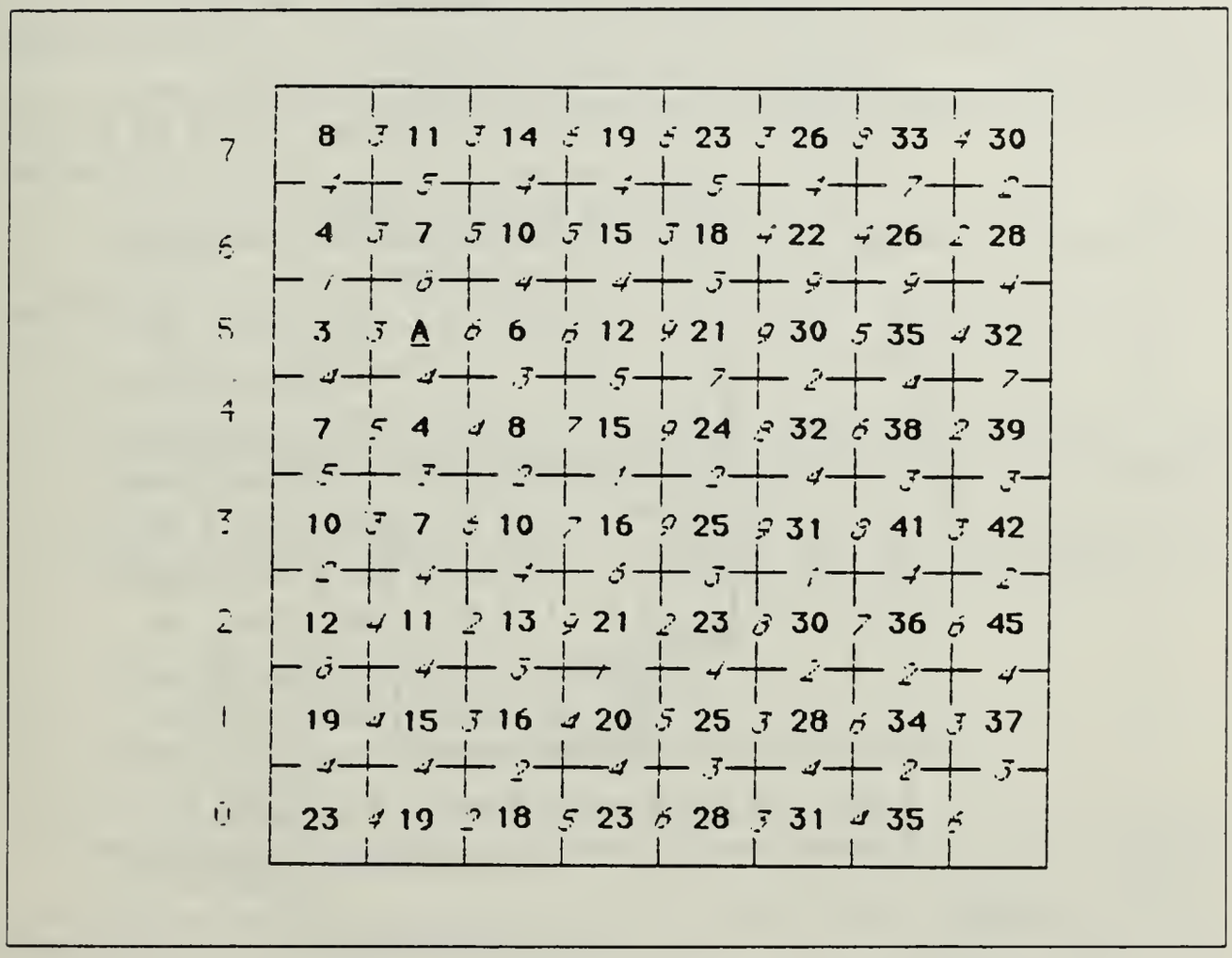

Figure 9: Cell values at the end of forward propagation.

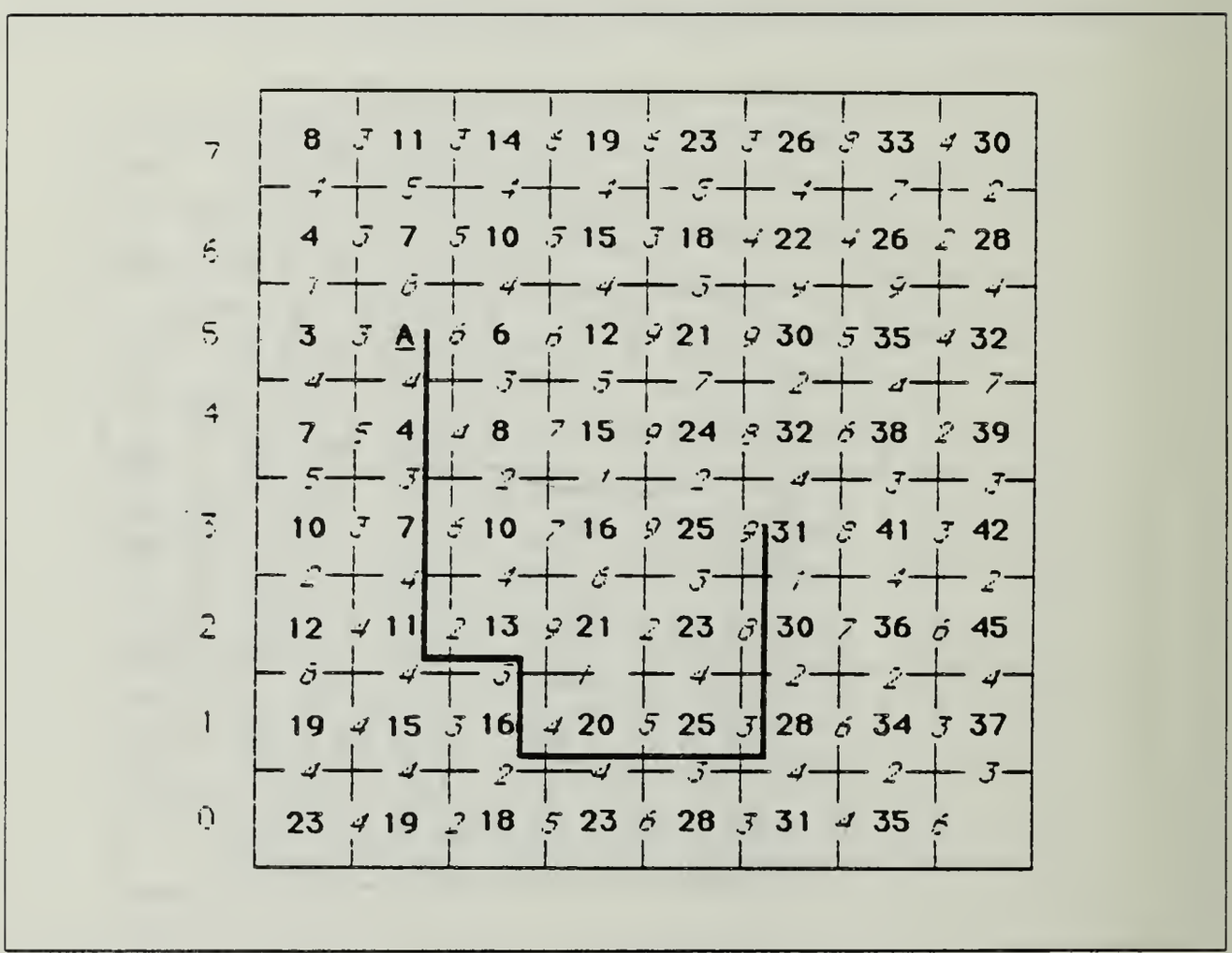

Figure 10: The backtrace route.

3. Channel Router

Channel routers are perhaps the most widely used routers today. Their popularity has resulted in the development of a wide collection of channel routing techniques that efficiently interconnect more diverse architectures. The first *channel router* was introduced in 1971 by Hashimoto and Stevens [Ref. 15] to route ILLIAC IV computer control boards. These two layer boards consisted of up to 165 identical dual-in-line IC packages in an 11 row by 15 column matrix. The channel router was developed to exploit the large regular routing regions created by this geometry.

As described by Rivest and Fiduccia [Ref. 16], a channel routing problem is specified by:

1. A channel length λ.
2. Top and bottom connection lists $T = (T_1, T_2, ..., T_\lambda)$ and $B = (B_1, B_2, ..., B_\lambda)$, where B_i and T_i are the net names for the pin at the ith column of the channel.
3. Left and right connection lists $L = (L_1, L_2, ...L_n)$ and $R = (R_1, R_2, ...R_n)$ specifying which nets extend through the ends of the routing channel.

A channel routing solution is specified by:

1. A channel width w representing the number of tracks used.
2. For each net n, a set of connected horizontal and vertical wire-segments whose endpoints are grid points (x, y) with $1 \leq y \leq w$ and $0 \leq x \leq \lambda$. Segments with endpoints $(i, 0)$ or $(i, w + 1)$, where i is the ith column of the channel, must be included in T_i or B_i. Nets in L must have endpoints at $x = 0$, while nets in R must have endpoints at $x = \lambda + 1$. Segments oriented along the channel length are called tracks and usually are assigned metal1, the layer with the best performance characteristics. Segments that run across the channel are usually shorter and are assigned a routing layer with inferior electrical properties, such as polysilicon. When required, the different layers are interconnected by means of cuts or vias.

To distinguish it from its descendants, the original channel router is commonly referred to as the *left-edge* channel router. It tries to minimize the channel width, w, by maximizing the number of nets sharing each track of the channel. This is done in the following manner. Nets are sorted by the location of their left endpoint.

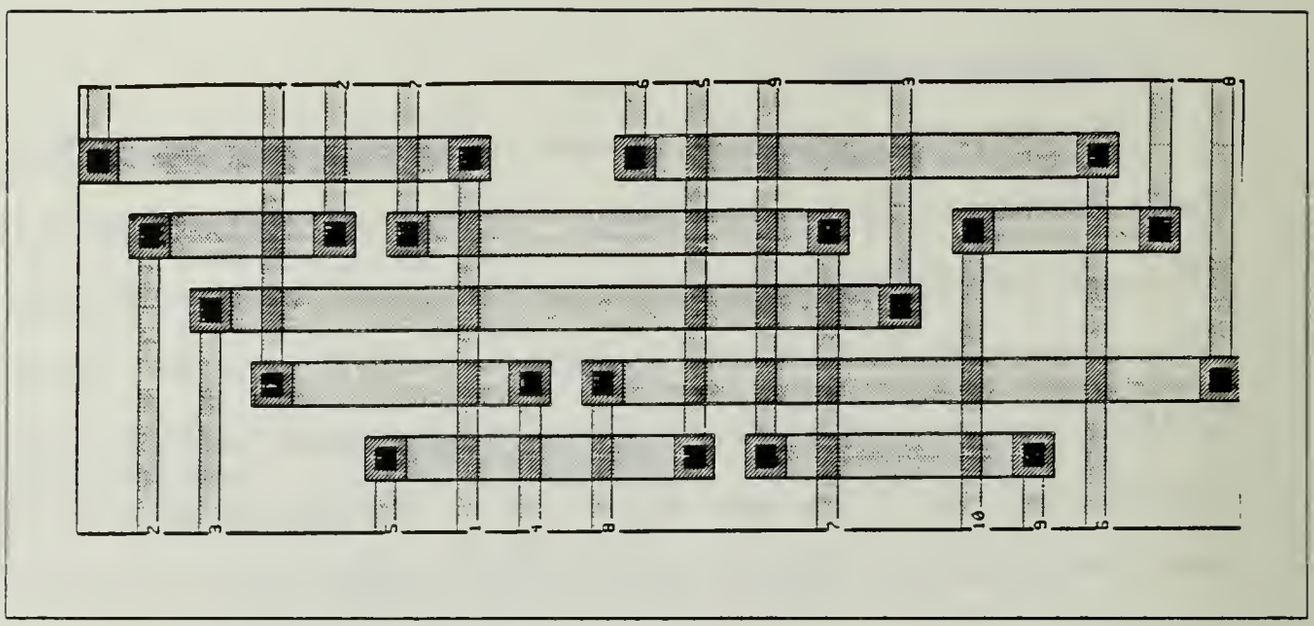

Figure 11: The left-edge channel router.

In the example on Figure 11, the sorted list is [1,2,3,4,5,6,7,8,9,10]. The router then selects the first endpoint, 1, and places it in the lowest left corner of the routing area. It next searches for a segment that does not overlap net 1. In this case, net 6. As nets are selected, they are removed from the sorted list. This process is repeated until no more nets can be placed in the first track. The algorithm then starts with track 2, which in this example is occupied by nets 2, 7 and 10. The process continues until all nets are assigned to a track. The track selections made by the left-edge algorithm are shown in Figure 11.

The track assignment method used by the left-edge channel router is guaranteed to find an optimal solution to a channel, provided no vertical constraints are present. If such constraints do exist, the left-edge router is not equipped to handle them. Vertical constraints occur when pins from different nets are placed opposite each other in a column. This forces one net above the other to avoid shorting the nets. Figure 12 shows a vertical constraints between nets 1 and 3, nets 1 and 2, nets 3 and 4, and others. Vertical constraints can be represented by a *vertical constraint graph*. A vertical constraint graph is a directed graph with its nodes represent-

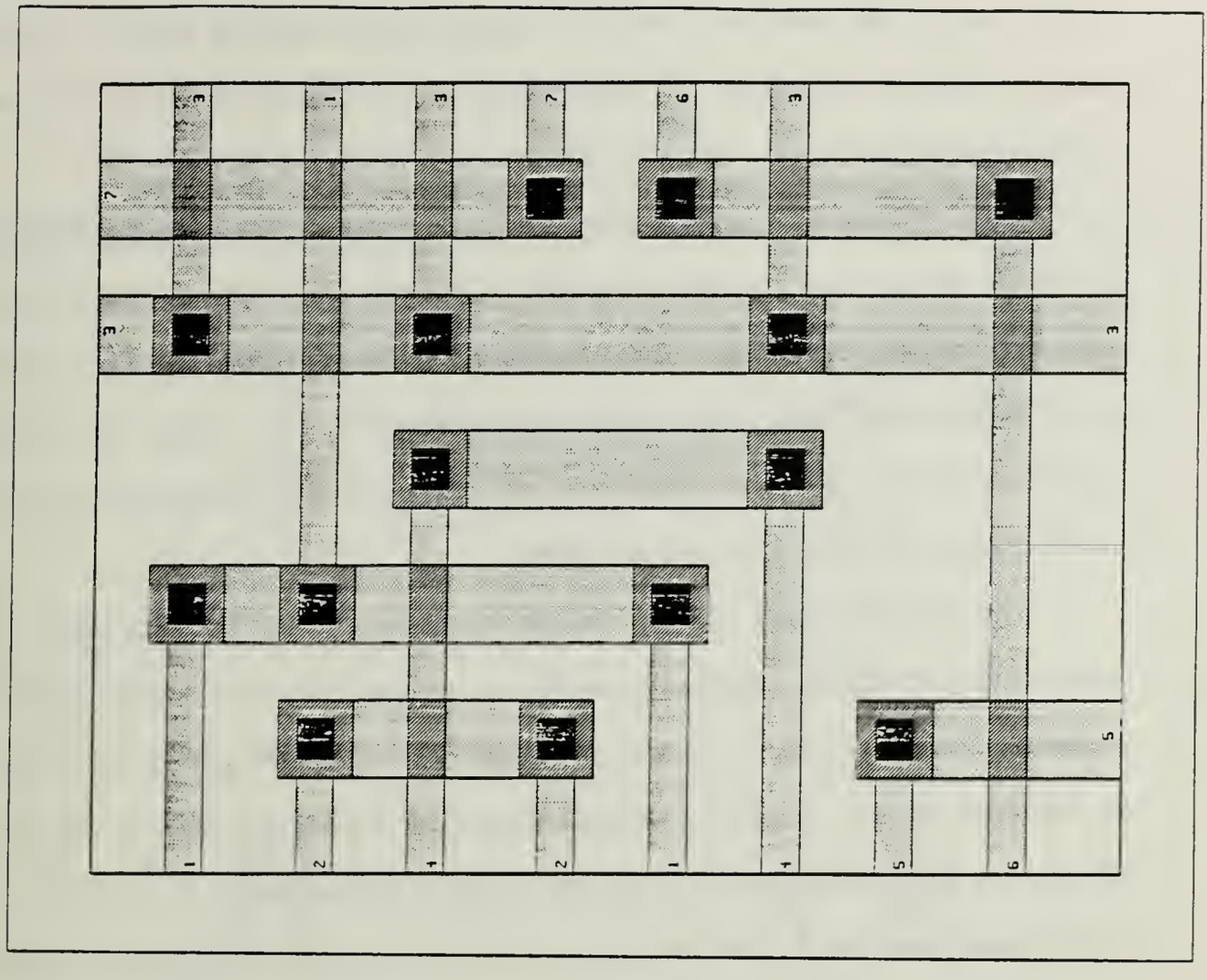

Figure 12: A channel with vertical constraints.

ing nets in the circuit and its branches depicting the relative position between the horizontal net segments in the channel.

Figure 13.d depicts the vertical constraint graph of the channel shown in Figure 12. The vertical constraint graph can be constructed iteratively by examining the nodes at each column of the channel. Starting at the left side of the channel described in Figure 12 net 3 is above net 1. This is represented on the vertical constraint graph by placing node 3 above node 1, and connecting them with a line as shown in Figure 13.a. In column 2, net 1 is above net 2. Since net 1 is already part of our graph, the line from node 1 is extended to node 2 as shown in Figure 13.b. In the next column net 3 is above net 4. The graph now looks as in Figure 13.c. This process continues until all columns have been examined. In the example,

this occurs at the right end with the column containing net 5 only. Since net 5 has no vertical constraints, it is represented by a single isolated node. The vertical constraint graph for the channel is illustrated in Figure 13.d.

The left-edge channel router fails to find a solution to channel problems with cycles in the vertical constraints graph. Cycles in the vertical constraint graph are called *vertical constraint loops*. As shown in Figure 14.a and 14.b, loops occur when there are at least two distinct paths connecting two nodes. As Figure 14.c shows, solutions to these problems require that one of the two nets be split into two horizontal paths. This is called *doglegging*.

The original *dogleg channel router* was proposed by Deutsch [Ref. 17]. It overcomes the vertical constraint problem presented above by breaking net 1 into subnets. Net 1a connects the top two pins and net 1b completes the connection to the bottom pin. The new vertical constraint graph is shown in Figure 14.d. Dividing a net into subnets is not always possible because nets can only be divided at columns where their pins are located. This means that cyclic loop problems can be solved provided the net has at least three terminals. Cyclic loops involving two terminal nets cannot be routed. Such a channel is illustrated in Figure 14.e.

Finally, a very different approach is taken by the *greedy channel router* proposed by Rivest and Fiduccia [Ref. 16]. Instead of routing the channel on a row by row basis as done by other channel routers, the greedy channel router routes one column at a time starting with the left edge. In each column it performs the following steps:

1. Makes connections to any pins at the top and bottom of the channel.
2. Combines as many as tracks as possible by merging or collapsing nets that currently occupy more than one track.
3. Reduces the distance between tracks occupied by nets still occupying more than one track.
4. Moves a net up if its next pin is on top of the channel or down if its next pin is on the bottom.

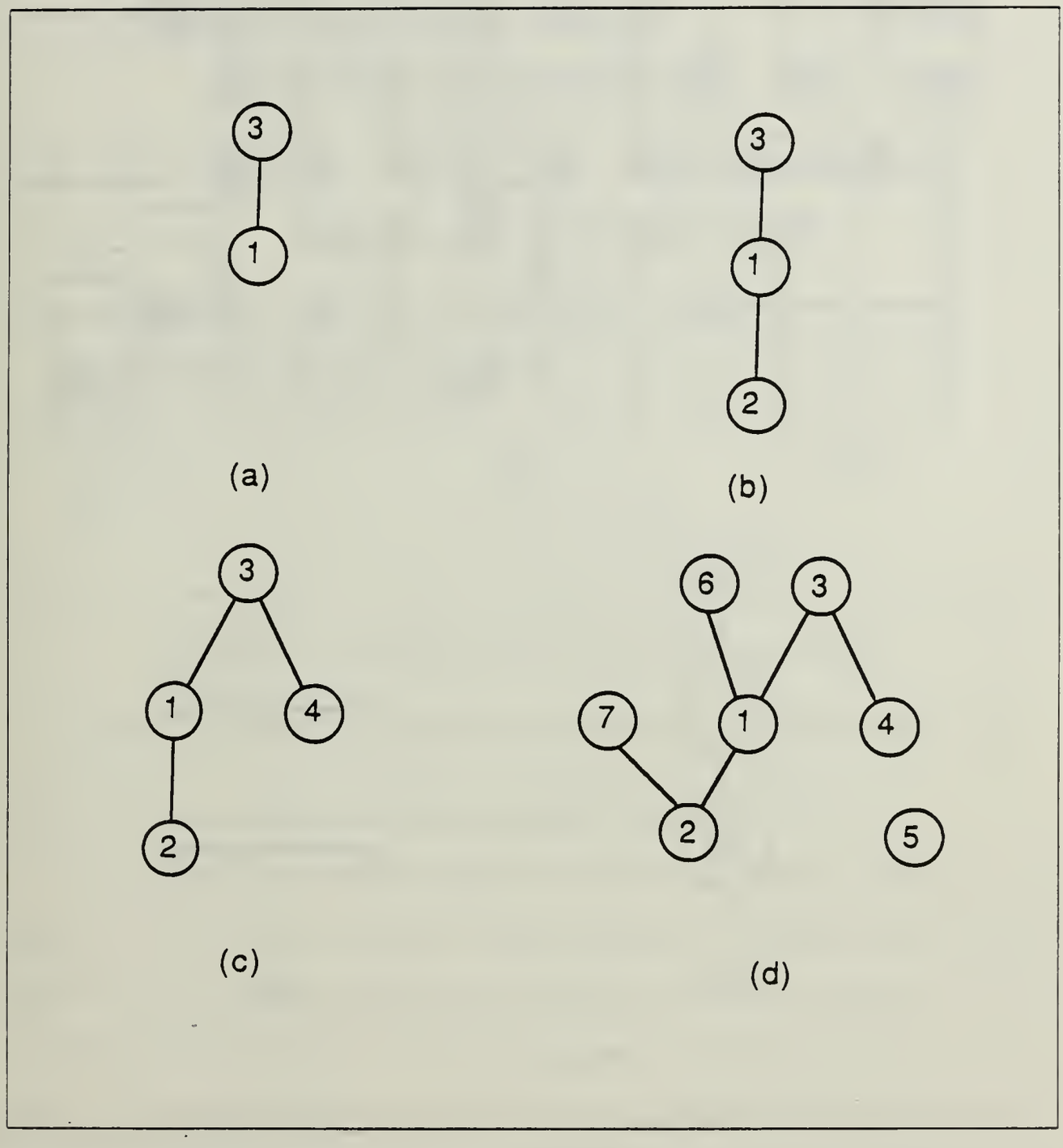

Figure 13: Making the vertical constraint graph for channel of Figure 12: (a) first column, (b) first two columns, (c) first three columns, (d) final solution.

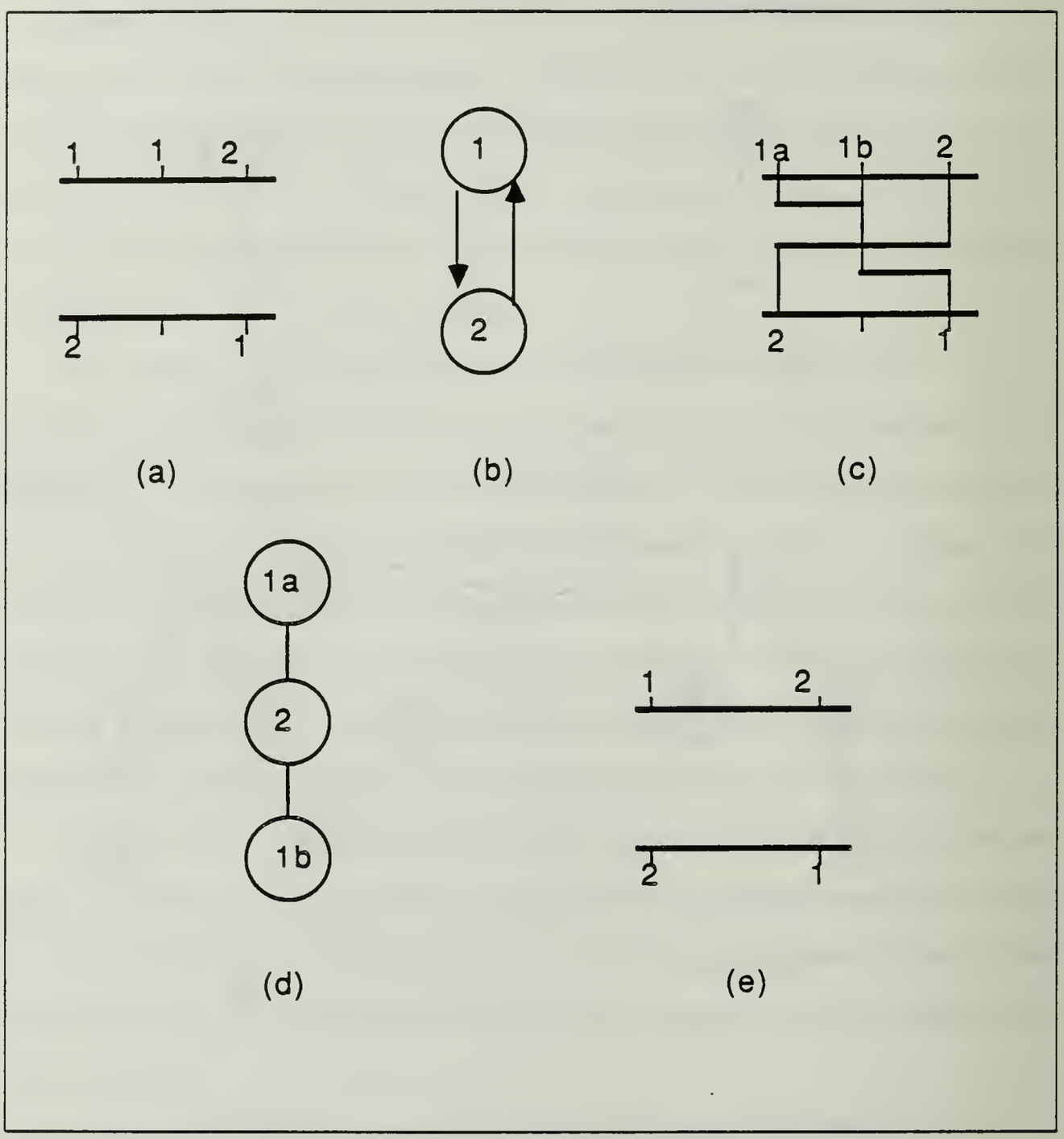

Figure 14: Vertical constraint loop: (a) channel, (b) graph of (a), (c) solution, (d) graph of (c), (e) channel that cannot be subdivided

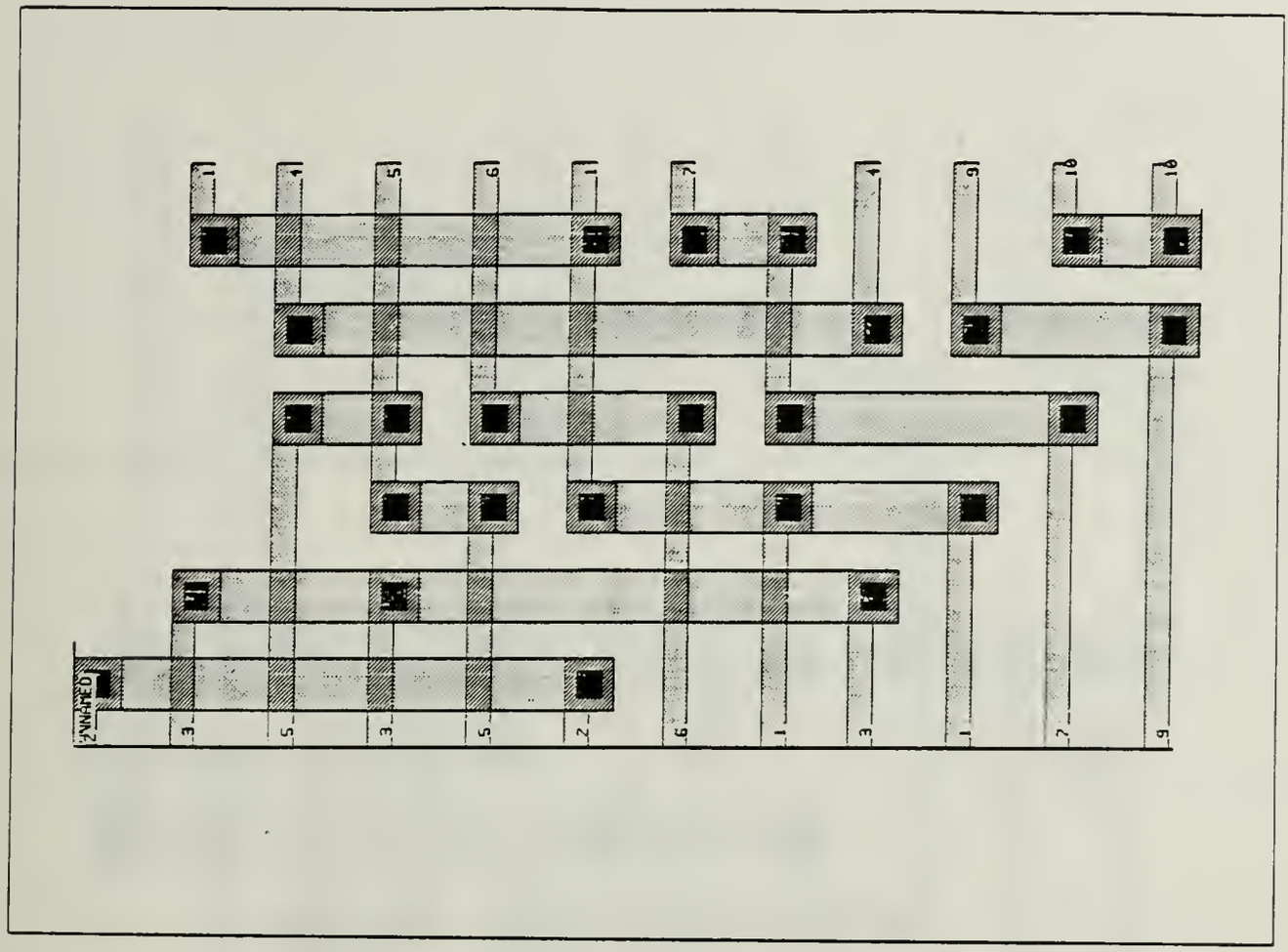

Figure 15: A channel routed by the greedy algorithm.

5. Adds a new track if a pin can't be connected because the channel is full.

Figure 15 shows a channel routed by the greedy algorithm.

The greedy algorithm still suffers from some of the problems afflicting other algorithms. First, since it looks for local optimums, conditions could be created that result in unworkable routing situations later on. Second, a cycle in the vertical constraint graph might make the channel impossible to route in the available channel length. The channel in Figure 16 provides such an example. Nets 3 and 10 cannot be completed because they are extended to the right edge of the channel. To complete this problem, the channel length must be increased by two columns.

4. River Router

The *river router* offers a fast and efficient exact embedding technique for the interconnection of a specific type of routing problem in one or more routing layers.

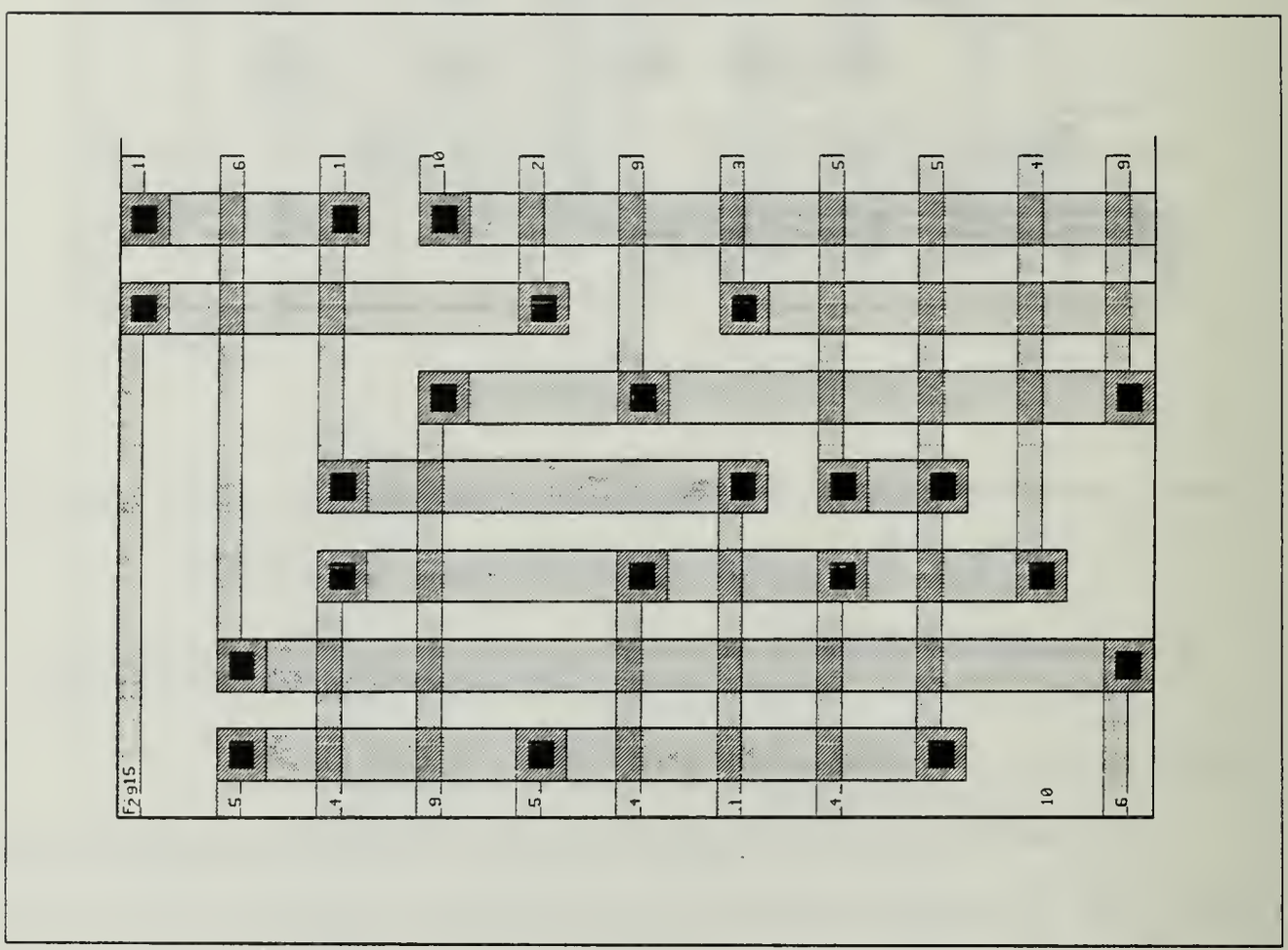

Figure 16: A channel unroutable by the greedy algorithm.

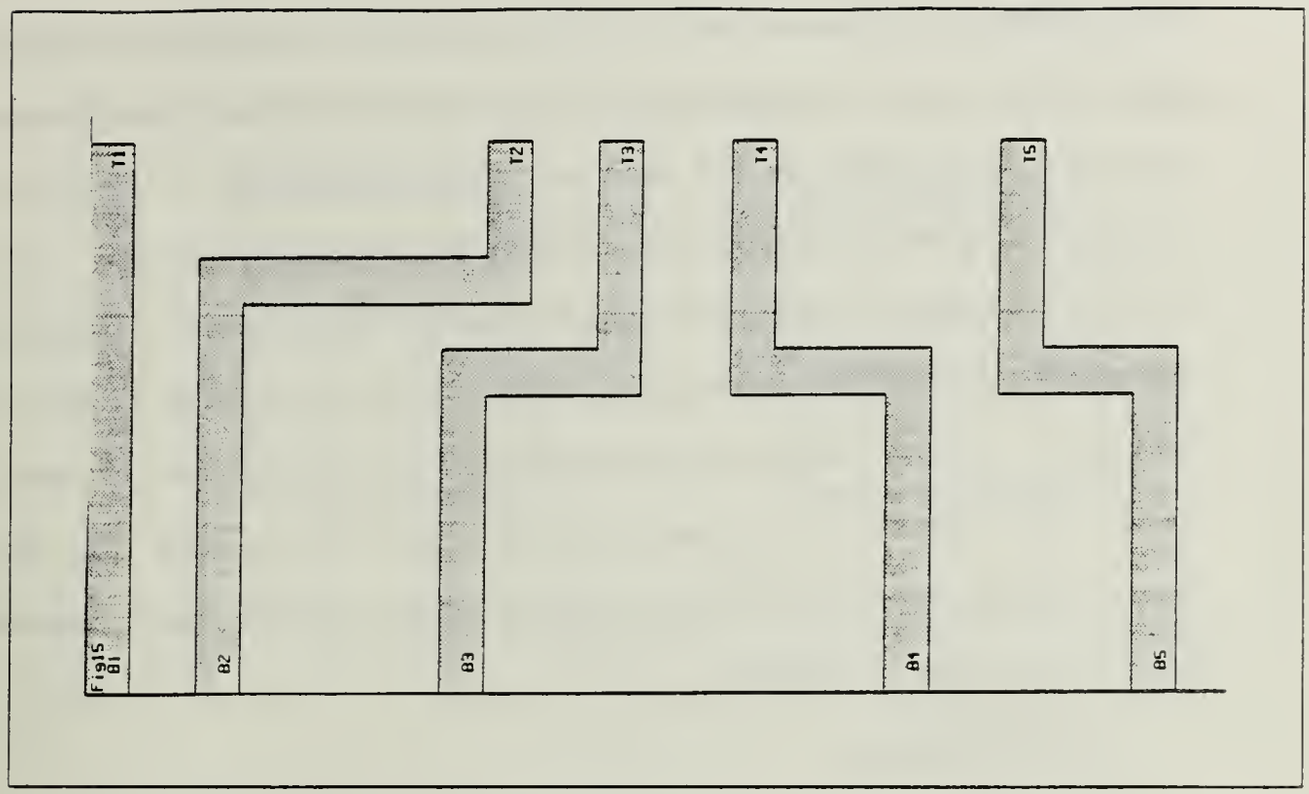

Figure 17: The river routing problem

It is the prescribed router when faced with:

1. A restricted or fixed routing space.
2. Terminals located on opposite sides of the routing area.
3. When only one routing layer is available, terminals are positioned so that no "crossovers" are necessary.

The router may be equipped with optimization routines to minimize the river width.

The river routing problem is more formally defined in the following manner. Given a rectangular routing region with fixed dimensions, a set of n connection points ordered from left to right along the top $T = (T_1, T_2, ..., T_n)$, and a set of n connection points ordered from left to right along the bottom $B = (B_1, B_2, ..., B_n)$. the river router generates n wires to interconnect each (B_i, T_i) pair (see Figure 17). The wires must lie within the fixed routing region. If a one-layer river router is used, no "crossovers" are permitted.

Although the river router solves the problem given to it efficiently, few circuits spontaneously produce layouts suitable for river routing. As a result, river routing

almost always presupposes less than optimal cell layouts as the order of pins in one river bank is made to correspond to the pin order across the river. These changes inevitably lead to less efficient designs. MacPitts' approach to routing between data-path and controller provides a clear example of this problem. To comply with the requirement for no crossovers imposed by the river router, MacPitts must compromise the optimization of either the data-path or controller layout. Since the data-path is usually the more complex structure, MacPitts decides to make the pin order of the controller correspond to the order in the data- path. The results are long horizontal polysilicon runs that span the entire controller, and a controller that may be larger than necessary.

5. Moat Router

The final step in the layout of an IC is the routing of internal cells to a ring of pads on the circuit periphery. The peculiarities of the pad routing problem generate a new set of heuristics which make the design of a routing scheme to specifically solve this problem worthwhile. Since the region to be routed is usually in the shape of a moat, the term moat router was coined by R. K. McGehee [Ref. 18].

The pad routing problem is actually an instance of the channel or river routing problem. In fact, the moat is really just a channel bent into a ring. The horizontal tracks found in the channel router are converted into concentric circles and vertical columns are transformed into radial columns (see Figure 18).

Moat routers can use one or more routing layers. As with channel routers, when two routing layers are available, the better conductor is assigned to wire the longer lengths. In the moat router this corresponds to the layers of concentric tracks. Since the radial columns are usually much shorter, the poorer conductor is used there.

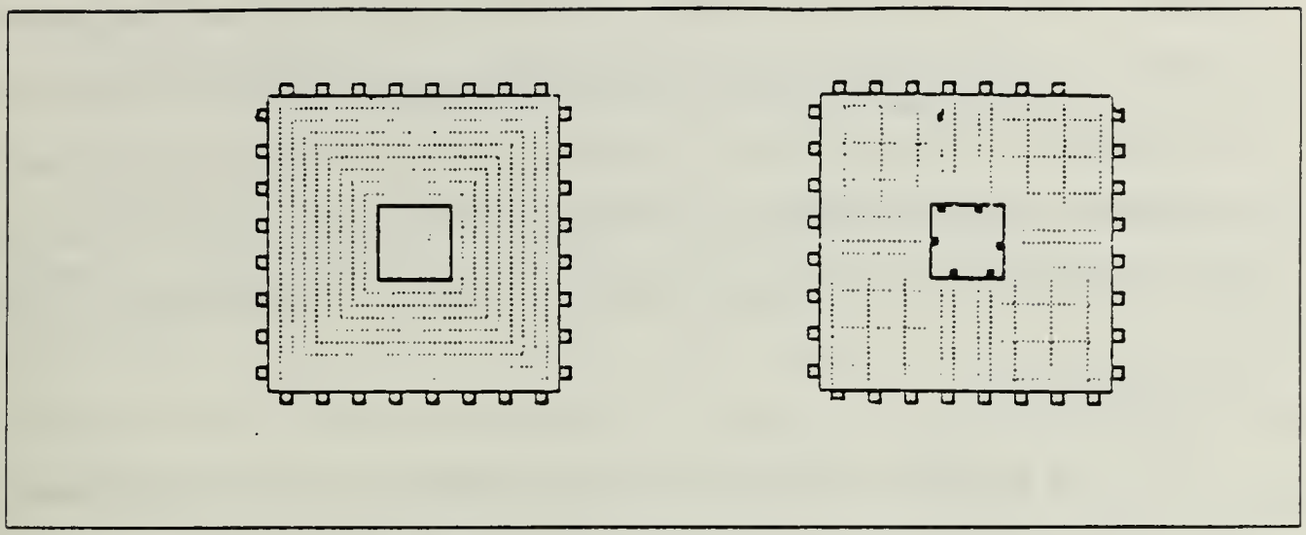

Figure 18: Concentric tracks and radial column geometry of moat routing region

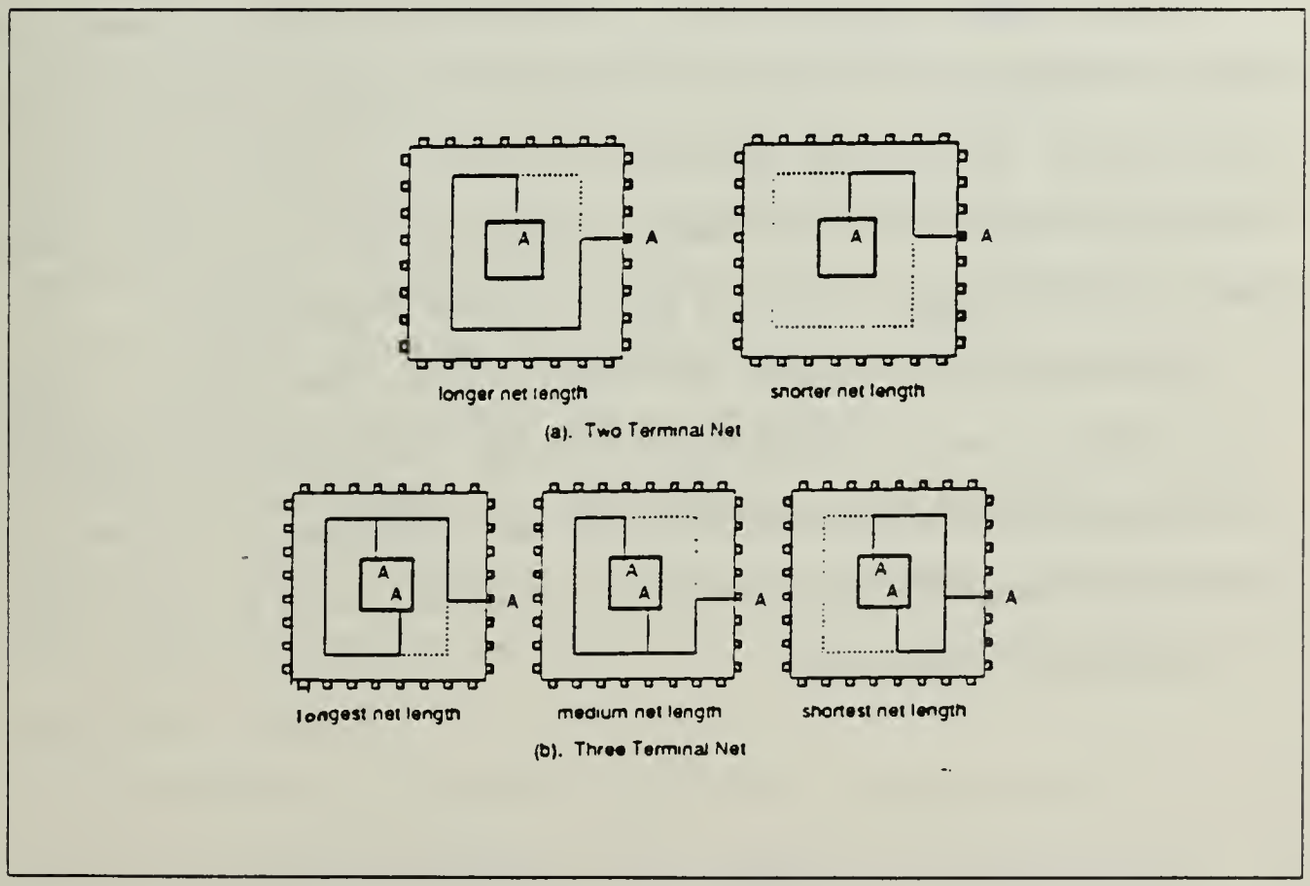

Figure 19: Moat routing direction ambiguity

Though similar in many respects to the channel and river routers, moat routers introduce a unique problem. Because a moat is a closed loop some ambiguity in selecting the best routing direction can occur. Within a channel there is no ambiguity in routing a net-list. In a moat, a two terminal net can be routed in two ways, a three terminal net can be routed in three ways, and so on (see Figure 19). Selecting the correct routing direction is crucial if optimal designs are desired.

C. SUMMARY

Selecting a specific router from the many available can be a very difficult problem. The global router divides a large problem into many smaller and simpler ones. The versatile Lee router is an exact embedding algorithm that will find an optimal route for a net, if a solution exists. Its demand for computer time and resources however, cannot be always satisfied.

Channel routers require much less information but are limited to those problems that can be expressed in terms of a well defined routing region with net connections at the edges. The left-edge router results in optimal solutions provided that no vertical constraints exist in the channel. Vertical constraints and cyclic constraints are handled efficiently by either the dogleg or greedy channel routers.

The moat and river routers are actually channel router variants. The moat router differs in that its routing areas form a ring. It is well suited to solve the pad routing problem. In the river router all nets have a terminal on each "river bank". When only one routing layer is available, the net-lists must be arranged so that no crossovers occur between nets.

III. ROUTING IN MACPITTS

The MacPitts compiling process produces a predictable design architecture consisting of a variety of functional and physical components. To completely interconnect a circuit, a variety of ad hoc routers, each intimately tied to the specific problem at hand, are used. In this section MacPitts' approach to routing between data-path units, between data-path and controller, and between the interior circuit body and pads is examined. The LISP functions discussed in this section are enclosed in Appendix A. Throughout this thesis, MacPitts' function names will be indicated by bold font and function arguments by italics.

A. ROUTING IN DATA-PATH

Starting with basic building blocks called organelles, MacPitts assembles a data-path hierarchically. The data-path can become very complex. An organelle is a hand-crafted one bit representation of a particular logic function, arithmetic function, shift function or comparison test. For an n-bit word, n organelles are stacked vertically to form a unit. A description of the MacPitts data-path design and routing organization is presented by E. Malagon [Ref. 9].

All inputs to and outputs from an organelle are routed to the data-path bus for transmission. Figure 20 illustrates a typical data-path design. Buses are located directly above the organelles in each bit slice. They are easily identifiable by the long polysilicon runs.

One curious feature of MacPitts designs is the requirement for all connections from the pads to enter through the left side. This has far reaching implications. For example, a signal produced at the far right end in the data-path must be routed

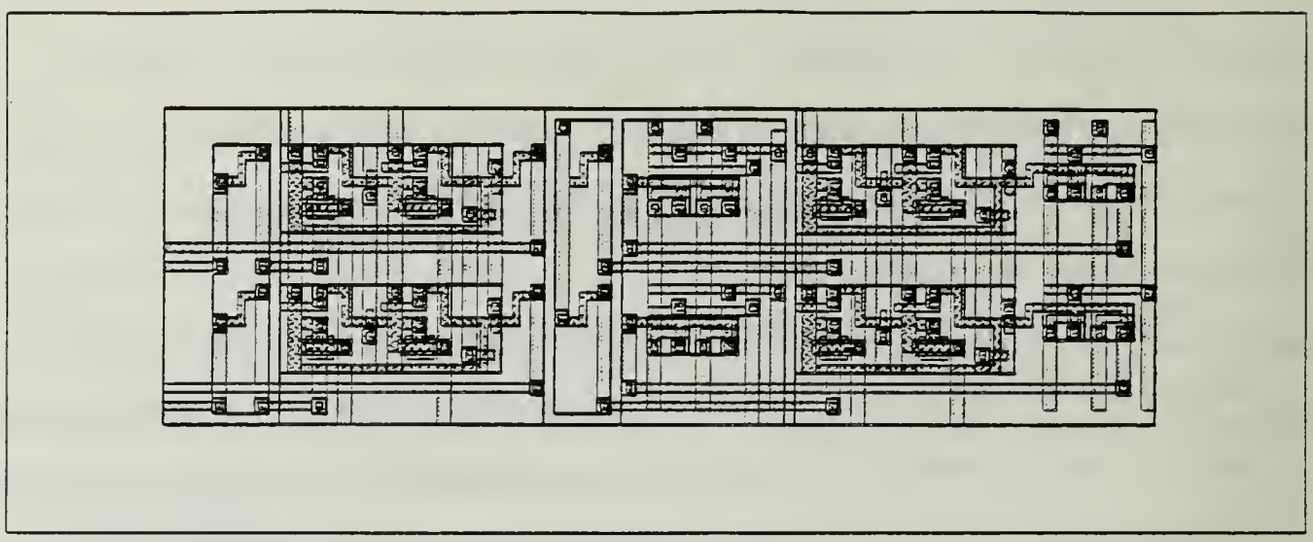

Figure 20: Typical MacPitts data-path design

across the entire data-path in polysilicon. For large circuits, polysilicon wires in excess of 1000λ are produced.

The data-path bus is produced by **get-buses-from-data-path**. This function calls on **get-buses-from-unit-lists** to collect all the participating bus terminals from the various data-path sources. The list of sources include: internal inputs, registers, multiplexers, output ports and organelles. The function that collects all the terminals with connections to pads is **get-basic-buses-from-port-output-unit**. This function uses the macro **make-left-tip** to insert the attribute 'left' to every port-output point. The left attribute causes the bus wire to extend from the internal connection to the left edge of data-path. If the macro **make-right-tip** were used instead, all data-path connections to the pads would take place through the right side.

B. ROUTING BETWEEN DATA-PATH AND CONTROLLER

MacPitts' control section performs Boolean logic operations on various signals to generate signals that drive the multiplexers in the data-path. The controller logic is obtained by means of a NOR gate array called a Weinberger array. Weinberger arrays offer advantages such as a relatively compact structure and no need to cross

signal nets. The controller connects directly to all control signal pads, tri-state pads, clock super-buffers and data-path.

Routing between data-path and controller, as well as between super-buffers and controller, is performed by a single layer river router with a non-optimal, but effective, channel-width calculation routine. The routing layer used in nMOS technology is polysilicon. This choice simplifies the routing algorithm and reduces area requirements (due to its smaller width). Unfortunately, when long wires are required, polysilicon's high resistance can slow the circuit down significantly.

On the positive side, the minimum feature size for polysilicon is less than that of metal, the only other practical alternative in nMOS technology. Each polysilicon track requires 4λ units, compared to 7λ units for metal1. A track is the sum of the minimum layer width and the minimum spacing between layers, as specified by the Mead-Conway design rules. A second advantage, relevant only in nMOS, is that using polysilicon eliminates two poly-metal cuts per net. These cuts would be required to cross over the power and ground rails located along both river banks. Since each cut contributes an average capacitance of 9.6×10^{-4} pf/mil^2 (based on metal over poly capacitance of 0.6×10^{-4} in 4μ nMOS technology [Ref. 1: Table 4.5, pg. 135]), the use of polysilicon, particularly in small circuits, may result in superior performance. This is of no consequence in SCMOS where two metal layers are available.

The problem with polysilicon is its relatively long signal propagation time delays. Polysilicon is approximately 100 times slower than metal1 [Ref. 1: Table 4.7, pg.136]. As routing material, it can be used without appreciable circuit performance degradation as long as $\tau_w \ll \tau_g$, where τ_w and τ_g refer to routing wire delays and gate delays, respectively. Under normal conditions, this relation holds true for polysilicon wire of lengths less than 200λ. Above 200λ, degraded circuit performance resulting from signal routing can be expected. Unfortunately, routing

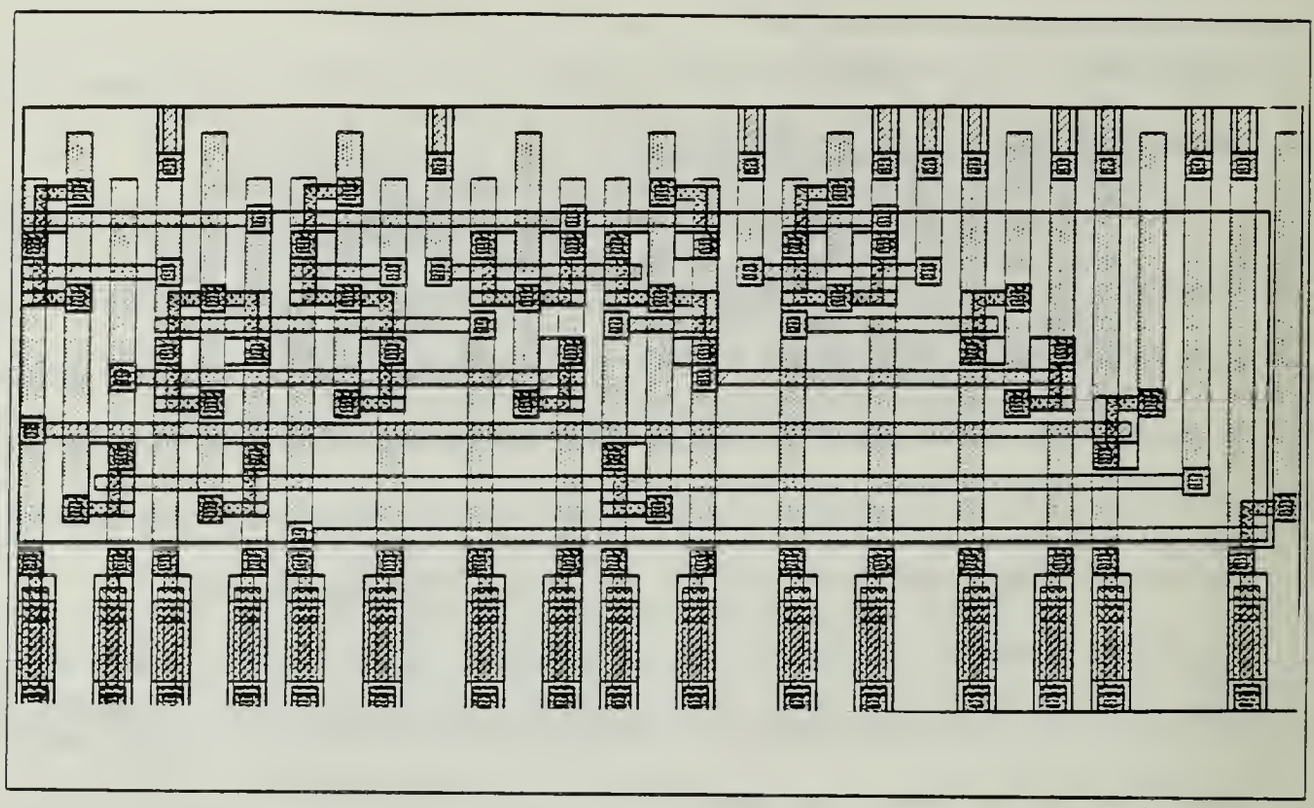

Figure 21: Typical MacPitts controller design.

lengths greater than 200λ are common in routing from data-path to controller. In fact lengths in excess of 500λ are found even in modest sized circuits.

Regardless of the layer used, the single most significant problem with the data-path to controller router is its use of a single routing layer. This means that, since crossovers are not allowed in the routing area, the terminal order in the controller must correspond exactly to that of the data-path. This is a serious design constraint that results in large, slow controllers. A typical MacPitts controller is illustrated in Figure 21. Notice the long horizontal polysilicon runs used to interconnect signals that interact with each other. Long runs could be avoided if the controller terminals were ordered to optimize the controller and not in response to the order of terminals in the data-path. Also, longer runs require more tracks, because less track sharing is possible. This increases the controller height and area. In summary, a one layer router has a negative impact on both controller performance and size.

In the process of laying out both the data-path and the controller, points with the 'river' attribute are created. A point is an L5 structure consisting of a point-name, x-coordinate, y-coordinate, layer and up to three attributes. L5 is the LISP-based layout language used by MacPitts. Attributes are keywords that either identify a point to a process, or give qualitative position information about that point. In this instance, 'river' identifies all interconnections between data-path and controller that need to participate in the river routing process.

The call to **river** originates in **layout-object**. Before calling **river**, layout-object first brings together all points that need to be interconnected into a net-list. It then formats this list as required by the river router.

Net-list extraction is accomplished by obtaining the desired information from larger lists. To this end, the variable *top-part*, which includes all details necessary to lay out both flags and data-path, and *bottom-part*, which contains the details to lay out the controller, are created. From these, the variables *top-bank* and *bottom-bank* are formed. They contain the x-coordinates, sorted in increasing order, of all points in *top-part* and *bottom-part* with the attribute 'river'. By design, the ith element in *top-bank* corresponds to the ith element in *bottom-bank*.

River is called in the following fashion:

(**river** *'NP 2* (**wing-span** *bottom-part) top- bank bottom-bank)*

River input parameters are: *layer, width, stretch, left* and *right*. In nMOS technology the *layer* NP indicates polysilicon. The *width* is 2λ, the minimum polysilicon width allowed by design rules. *Stretch* will be discussed momentarily. *Left* and *right* correspond to *top-bank* and *bottom-bank*, respectively. This change in names, top to left and bottom to right, reflects an actual change of orientation that occurs while in **river**. The original orientation is recovered by rotating the output list, *river-layout*, clockwise before appending it to *internal-layout*. *Internal-layout* contains

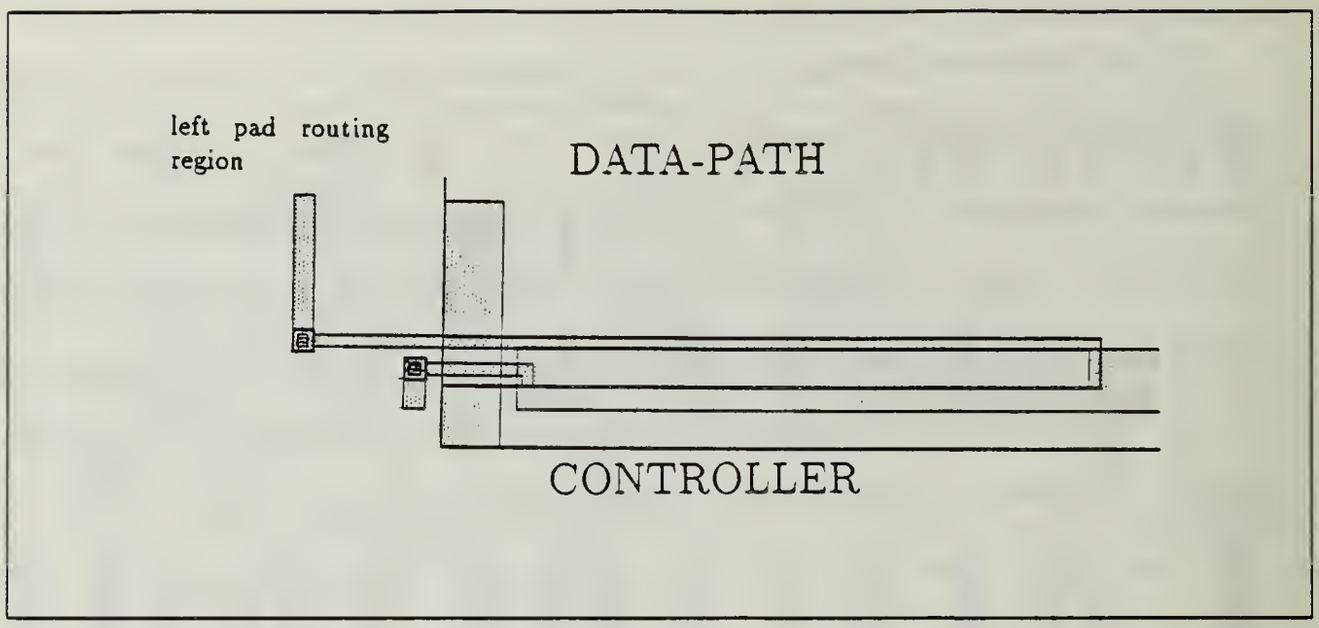

Figure 22: Wing layout

everything necessary to produce a layout of the circuit body: *top-part, bottom-part, wing-layout river-layout* and *skeleton.*

Stretch represents the height of *wing-layout.* The name reminds us how far each net must 'stretch' past the bottom river-bank to reach its terminal on the controller. It is determined by calling on **wing-span** with *bottom-part* as an argument. **Wing-span** extracts all points with the 'wing' attribute from *bottom-part.* The attribute 'wing' identifies those control signals from the controller that connect to pads. For each such point, **wing-span** increments stretch by 5λ. Finally, an additional 1λ is tacked on to stretch. Why does **wing-span** require 5λ between tracks while **river** needs only 4λ? As illustrated in Figure 22, since each 'wing' net terminates on a poly-metal cut during pad routing, the extra 1λ is necessary to satisfy design rule requirements for 2λ separation between poly and poly-metal cuts in the left pad routing region.

The river routing process is performed by **river, river1, river-span** and **river-span1**. As their names imply, **river-span** and **river-span1** calculate the required river width to accomodate all nets. Although not an optimal algorithm, **river-span** usually finds optimal or near optimal solutions. **River-span** uses *layer, space, left*

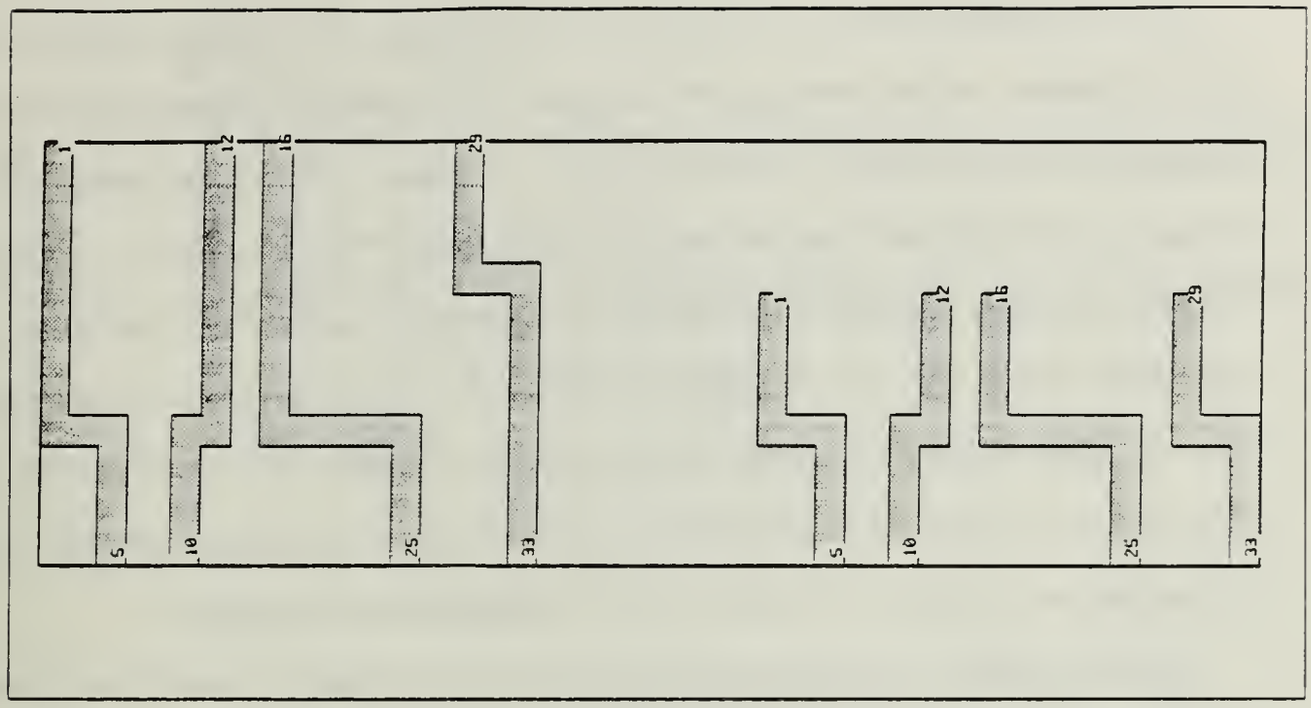

Figure 23: Solution to river routing problem, (a) MacPitts solution, (b) optimal solution.

and right as parameters. It determines the number of tracks required in the channel by calculating the run-length — the maximum number of consecutive nets for which the x-coordinate of one bank is greater than that of the other. For example, a net-list can be described by ((1 12 16 29) (5 10 25 33)). The list enclosed in the first inner parentheses represents the net terminals on the top-bank, while the list in the second parentheses corresponds to net terminals on the bottom bank. For this net-list **river-span** returns a run-length value of 2 because this is the number of consecutive nets where the terminals on one bank are consistently either greater than or less than the terminals on the other bank. In this case the nets are (20 30) $\rightarrow$ (40 50). The product of the run length and 4λ, the minimum space required per polysilicon track, gives the span of the river. The algorithm is not optimal because it fails to recognize that nets which meet this condition and do not overlap can share the same track. This illustrated in Figure 23. MacPitts' solution to the problem (Figure 23.a) uses two tracks. The optimal solution, shown in Figure 23.b requires only one track.

The real workhorse in the river router is **river1**. To function, **river1** needs the arguments used by **river** as well as *span, where* and *flag. Span* is the distance calculated by **river-span**. *Where* is the y-coordinate of the track which the net being routed is to use. *Flag* can take as values either "down", "up" or "straight". "Down" indicates routing in the downriver direction $T_i > B_i$. "Up" indicates routing upriver when $B_i > T_i$. "Straight" corresponds to routing across the river when $T_i = B_i$. Essentially, *flag* provides a means for the algorithm to 'remember' whether the previous routed net was routed upriver, downriver or straight across. *Flag* is important to determine the value of *where* during the next iteration.

River1 employs a simple and effective recursive routine to route all nets between *left* and *right* in the manner shown in Figure 23.a. Figure 23.b illustrates the same problem optimally routed. Since the nets are pre-sorted, applying the basic LISP function 'car' to both top and bottom-banks yields the terminals that need to be routed in the current net. Applying 'cdr' simultaneously to *top-bank* and *bottom-bank* exposes the next net.

River1 first compares the x-coordinates of the ith net by applying the operators $=, >$, and $<$ to the ith elements in *left* and *right*. The following listing illustrates all the possible cases.

1. IF $left_i = right_i$? THEN
    * A single vertical polysilicon wire segment is laid out between net terminals.
    * $flag =$ '*straight*'

2. IF $left_i > right_i$? AND
    - IF $flag =$ '*down*'? THEN
        * A vertical *layer* segment is laid down from data-path to *where*.
        * A horizontal *layer* is laid down from *left* to *right*.
        * A vertical *layer* is laid down from *where* to the controller terminal.
        * $where = where + (width + space)$.
        * $flag =$ '*down*'

- IF *flag* = '*up*'? THEN

    * A vertical *layer* segment is laid down from data-path to the first track.
    * A horizontal *layer* is laid down from *left* to *right.*
    * A vertical *layer* is laid down from the first track to the controller terminal.
    * $where = where + (width + space)$.
    * $flag =$ '*down*'

3. IF $left_i < right_i$? AND

    - IF *flag* = '*up*'? THEN

        * A vertical *layer* segment is laid down from data-path to *where.*
        * A horizontal *layer* is laid down from *left* to *right.*
        * A vertical *layer* is laid down from *where* to the controller terminal.
        * $where = where - (width + space)$.
        * $flag =$ '*up*'

    - IF *flag* = '*down*'? THEN

        * A vertical *layer* segment is laid down from data- path to the last track.
        * A horizontal *layer* is laid down from *left* to *right.*
        * A vertical *layer* is laid down from the last track to the controller terminal.
        * $where = where - (width + space)$.
        * $flag =$ '*up*'

C. ROUTING TO PADS

There are three phases to the process of routing from the circuit body to the pads. They are pad placement, net extraction and net layout. MacPitts' approach to all three is very inefficient. There are three significant flaws with the layout method used by MacPitts. First, since all inputs into the circuit body must enter through the left side, extremely long routes and unecessarily wide channels are

formed. This problem is not caused by the pad router functions, but by the assignment of the 'left' attribute by data-path, to all pad connections. A detailed explanation is given in section III.A. Second, pad placement is only allowed on the top, right and bottom sides. If the pads do not fit, the chip is extended in length or width or both until all pads can be accomodated on those three sides. Finally, pad position is dependent on the pad number assigned by the user in the source file and not on any optimizing algorithm. The combined effects of these problems are the large empty spaces usually found in circuits designed by MacPitts. This is illustrated in Figure 24.

1. Pad Placement

MacPitts' pad placement algorithm lacks any 'intelligence' to improve either circuit performance or area utilization. The functions directly involved in this process are **place-pins, extend-right** and **extend-top**.

Interestingly enough, **place-pins** is capable of placing pins on all four sides. It uses a four case conditional to assign pads with pin numbers less than *number-pins-per-side* to the top; it then assigns those pads with numbers less than twice *number- pins-per-sides* to the right side; next, pin numbers less than three times *number-pins-per-sides* are assigned to the bottom. Any remaining pads are assigned to the left side. Pin numbers are assigned to pads in the circuit source file.

Why then doesn't MacPitts place pads on all four sides? Because *number-pins-per-sides* is calculated by dividing the total number of pads by three and then rounding the result up to the next integer. Thus, circuits with 16, 17 and 18 pads all yield 6 as a value for *number-pins-per-side.* By the time **place-pins** gets to the left side, there are no pads left to place.

The functions **extend-right** and **extend-top** are used if the current circuit length or width is insufficient to accomodate the number of pins assigned to it. These functions are independent of each other, responding only to requirements

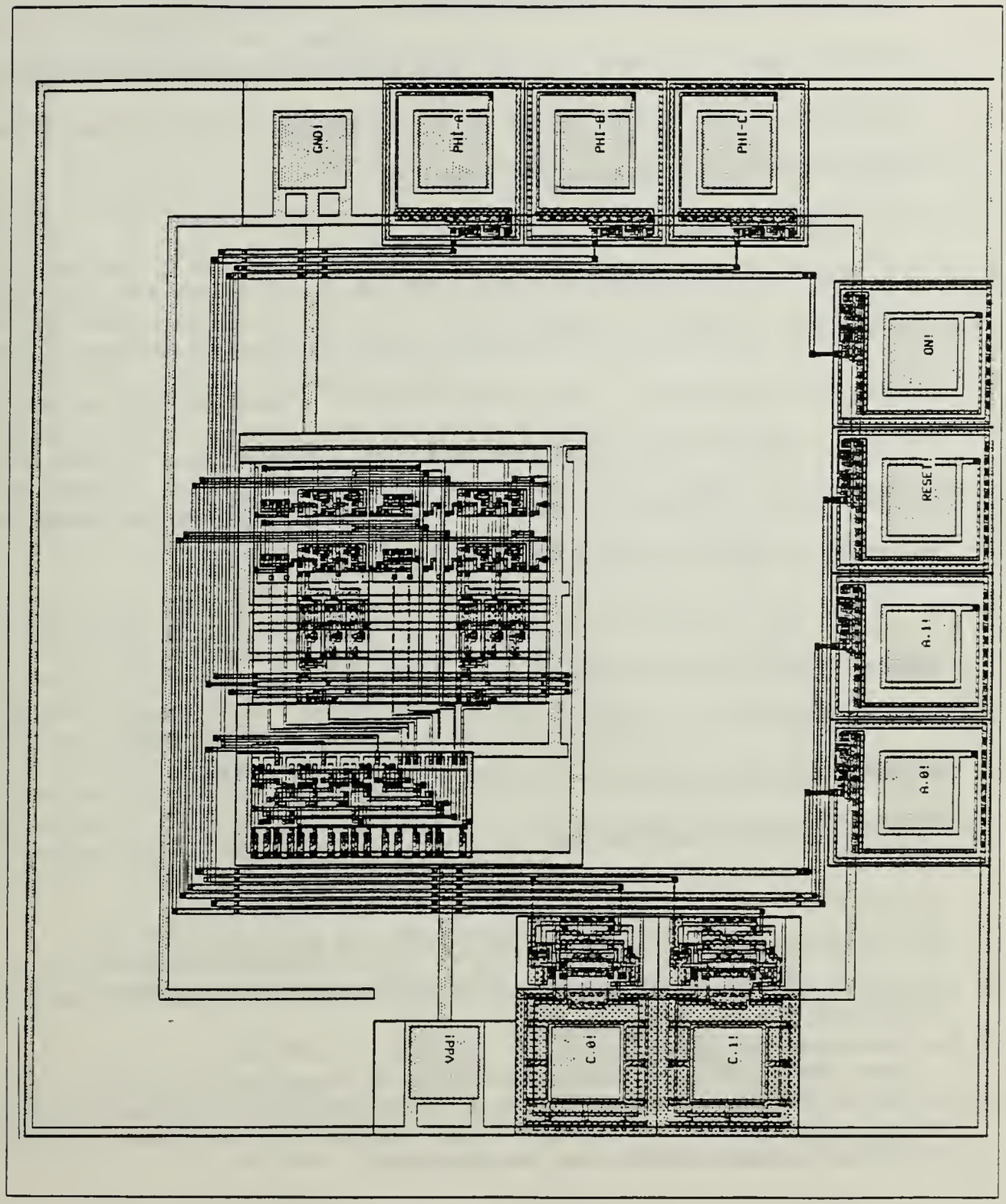

Figure 24: Routing pads in MacPitts

set by *number-pins-per-side*. For example, an 18 pad circuit with a length that accomodates 10 pads and a width that fits only one will be extended by **extend-top** until the six pads required by *number-pins-per-side* fit on each side; a five-fold increase in area. The fact that the original dimensions can accomodate 22 pads is irrelevant. These extensions can have a drastic impact on the final chip size as demonstrated by this example and Figure 24.

2. Net Extraction

The functions directly involved in the extraction of nets are: **extract-nets, extract-basic-nets, order-basic-nets, rotate-basic-nets, extract-subnets** and **extract-[side]-subnet.** [1] All the data necessary to form net-lists can be found in the parameters *pins-layout* and *internal-layout*. *Internal-layout* contains every detail necessary to actually lay down the circuit body. Likewise, *pins-layout* contains everything needed to produce a layout of the pads.

MacPitts starts the net extraction process by merging *internal-layout* to *pins-layout*. This list is then operated on by **extract-basic-nets**. This function serves two purposes. First, it makes a list of lists, with the inner lists containing all points with the same name. Second, it retains only those points with the attribute 'ring'. The ring attribute identifies those terminals that participate in pad routing. To an input such as:

(((1 1) 10 20 nil (nil nil nil)) ((port input (a 1)) 0 100 nil (ring left nil))
((port input (a 1)) 100 200 nil (ring right nil)) ((phic) 0 50 nil (nil left ring))
((phic) 100 200 nil (nil top ring)))

[1] Many functions in the net extraction and net layout business come in four flavors: top, right, bottom and left. For each function type all flavors operate in the same fashion. For clarity and brevity, the generic [side] will be inserted in place of the specific flavor whenever the clarity of the issue being discussed won't suffer from the substitution.

extract-basic-nets returns the following list:

((((port input (a 1)) 0 100 nil (ring left nil))
((port input (a 1)) 100 200 nil (ring right nil)))
(((phic) 0 50 nil (nil left ring))
((phic) 100 200 nil (nil top ring))))

Next, each net in basic-nets is arranged in proper order by **order-basic-nets**. This is accomplished by sorting each net with respect to an operator supplied by **basic-net-point-further-left?**. This function examines the attributes of each of the two points and supplies the correct sorting criteria to handle the specific problem. For example, given the net:
(((phic) 0 50 nil (nil left ring)) ((phic) 100 200 nil (nil top ring)))
the operator (< (point-y point1) (point-y point2)) is supplied. In this case point-y of point1 and point2 equal 50 and 200, respectively, so this net is already in the correct order.

At this juncture each net consists of at least two ordered points. The nets are now processed by a rudimentary global router. It is rudimentary because, while it does assign routing regions to each net, it does so with little emphasis on optimization. This global routing function is the responsibility of **extract-subnets, extract-subnet** and **extract-[side]-subnet.**

These functions work in a straightforward fashion. Remember that each net point may have up to three attributes. The set of allowable attributes depends on the operation that the point is destined for. For example, all points involved in pad routing have the attribute 'ring'. A second attribute identifies the side where the point lies and can take values of either 'top', 'right', 'bottom' or 'left'. The third attribute gives additional position information and can take values of 'first', 'last' or 'nil'. **Extract-subnet** queries the first point of a net to determine which side it is on. It does this through the macros **is-point-top?, is-point-right?, is-point-bottom?** or **is-point-left?**. The net point must respond with true to

one of the four macros. That macro will in turn call **extract-[side]-subnet**, where [side] is the side in the point's attribute list. For example, if the first point is on 'top', **extract-top-subnet** is called.

The various **extract-[side]-subnet** functions supply points necessary to allow continuous routing of a net from source to target. Continuity cannot be guaranteed for a two point net. The net called (port input (a 1)) of the previous example illustrates this point. One point lies on the right side and the other on the left. To connect them, **extract-[side]-subnet** must generate four new points. Two directions are possible; either through the 'top' or the 'bottom'. Assume that in this instance the top is a shorter route and is therefore selected. The new net is now:

```
(((port input (a 1)) nil 0 100 (ring left nil))
((port input (a 1)) nil 100 200 (ring right nil))
((port input (a 1) nil nil nil (ring left last))
((port input (a 1)) nil nil nil (ring top first))
((port input (a 1)) nil nil nil (ring top last))
((port input (a 1)) nil nil nil (ring right last)))
```

If the original two points were on a different set of sides than the example above, a different set of points would be created. The process, however, remains the same.

Extract-subnets can operate on nets with more than two terminals on one or more sides. This is possible because **extract-[side]-subnet** determines the side of the next point in its original net, and calls on the corresponding **extract-[side]-subnet**. The process continues until all points in the net have been examined. This capability is used in routing the control signal that places tri-state pads in either the high or low impedance mode.

The final phase in the extraction process is the allocation of a track to each net segment. While MacPitts' track allocation algorithm optimizes the channel width, its inability to consider anything but the current channel results in a less than optimal overall layout. It lacks the intelligence to assign tracks to nets so

that cuts and poly bridges are not required at the channel corners and between pad-terminals and their tracks.

3. Net Layout

The output from the net extraction process is called *nets*. It is used by **layout-nets, layout-[side]-net** and **layout-[side]-point** to perform the detailed routing. The pad router uses two layers, metal1 and polysilicon to route the nets. Metal1 is the preferred layer due to its superior performance characteristics. Polysilicon is always used along with two poly-metal cuts to cross-over metal interferences from either power/ground metal1 lines or other nets. Henceforth, this structure will be referred to as a poly-bridge.

Layout-nets is inefficient code. It will insert poly-bridges at the pad terminals and at the corners between routing areas even when this costly structure is not required. This is illustrated in Figure 24. Notice how a poly-bridge is used in connecting segments at the top-right corner when continuous routing in metal1 is possible. In other words, the algorithm always routes for the worst case.

Layout-nets routes one net at a time. To assemble a layout it considers three variables: the side of the current segment, its endpoints and the location of the power and ground pads.

The side where the current point is located is found by use of the macros **is-point-[side]?**. A 'true' condition is returned for the macro whose [side] matches the side attribute of the point being investigated. Consequently, **layout-[side]-nets**, where [side] is the same as the side in the point attribute list, is called with the net, to which the point belongs, as an argument.

In addition to *net*, the parameters *top, right, skip* and *power* are also used by **layout-[side]-net**. *Top* and *right* are the width and length dimensions of the circuit body. They correspond to *extended-top* and *extended-right* in Figure 25. *Skip* is the x-coordinate of either the power or ground terminal, whichever is on the same side

as the net point in question. If neither power nor ground is on the point's side, then *skip* is given nil as a value. Finally, *power* is the width of the metal wire connecting ground or power to the power/ground skeleton. *Skip* and *power* are used to position poly-bridges and allow nets to cross-over power/ground wires.

In laying out a net segment, **layout-[side]-net** considers all the points of a net lying on that [side]. The function calculates the coordinates necessary to extend the layer from one point to the other. The actual instrument used to lay out the wire is the L5 function **rect**. As the name suggests, **rect** defines the boundaries of a rectangle in a specified layer. Its arguments are: *layer, x-min, y-min, x-max* and *y-max*. Information gleaned from the attributes ('first', 'last' or 'nil'), the track number assigned to each side of a net by **allocate-tracks**, and the design rule specifications are used to determine the value of the arguments used by **rect**.

Layout-[side]-net determines the segment endpoints from position attributes of the net points on that side. It uses the macros **is-point-first?** and **is-point-last?** to ascertain if either 'first' or 'last' occur in the point attribute list. If the attribute is 'first', while operating on either the bottom or top sides, then a value of 2 is assigned to *x-min*. If operating on either the left or right side, then *y-max* takes the value *top*. If the attribute were 'last' then *x-max* is given $(right - 2)$ for a value when operating on either the top or bottom sides, or $y\text{-}min = 0$ when on the left or right sides. If neither 'first' nor 'last' is in the attribute list, then the point must be either a pad or an internal terminal. The correct endpoint is obtained by using either the macro **point-x** when routing on the top or bottom sides, or **point-y** when routing on the left or right sides.

The method described in the previous paragraph finds the segment endpoints along the routing channel. How are coordinates specifying the segment's location across the channel obtained? The location of the routing channels on the chip is known. The lower-left corner of the circuit body lies on the point given by the

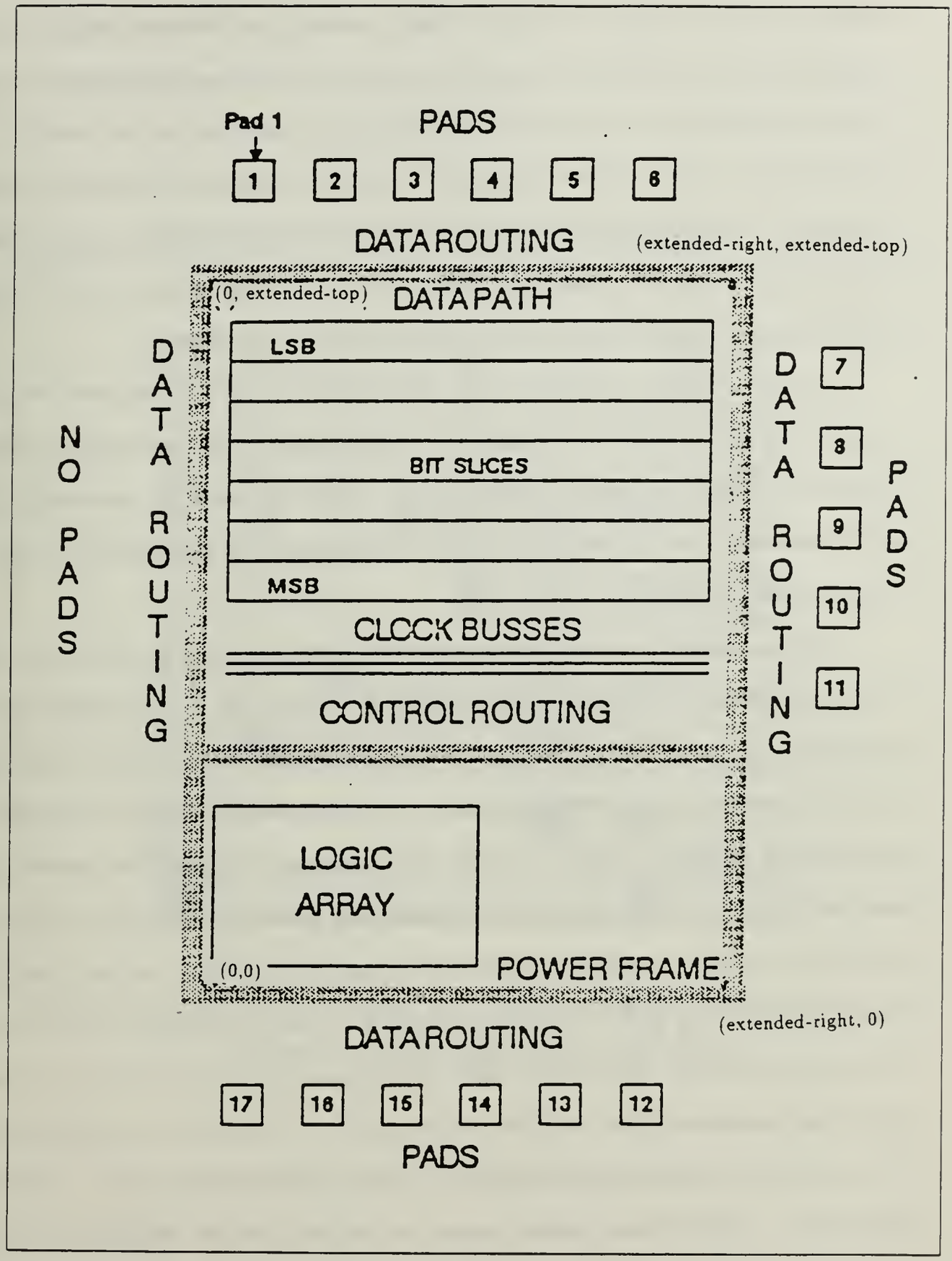

Figure 25: MacPitts floor plan.

Cartesian coordinates $(0, 0)$. The top-right corner, on the other hand lies on the point $(right, top)$. These two points define a rectangular reference frame which is simultaneously the exterior boundary to the circuit body and the inner boundary to the routing areas. The tracks are evenly spaced concentric circles, starting with track number 1 near the inner boundary. The space between tracks and the width of each wire is obtained from the design rules. With this information, **layout-[side]-net** is able to obtain the missing y-axis value when routing on either top or bottom sides, or x-axis values when routing on either right or left sides.

So far, the pad routing area has been visualized as four disjoint rectangles surrounding a circuit body. Actually, the routing area is a rectangular ring consisting of four disjoint rectangular areas and four small, square areas connecting them at the corners. The interconnection of net segments at the ring corners is done by **layout-[side]-point**.

Layout-[side]-net calls on **layout-[side]-point**, where in both cases [side] is the same as the side of the net segment being operated on. **Layout-[side]-point** considers three questions to determine the correct layout: which corner? from which track? and to which track? The answer to the first question hinges on whether the point has an attribute of 'first' or 'last'. For example, if currently in **layout-top-net** with 'first' as an attribute, then **layout-top-point** is called with "left" as a parameter; meaning, connect top and left segments. Likewise, the attribute 'last' calls on **layout-top-point** with the parameter "right", indicating a connection at the top-right corner. The current point track number is obtained with the macro **net-track-number**. The track number of the point it connects to is obtained with the macros **last-point-track-number** or **first-point-track-number**. With this information, **layout-[side]-point** is able to connect two net segments as shown in Figure 26. Notice that regardless of the need for a poly-bridge, one is always created.

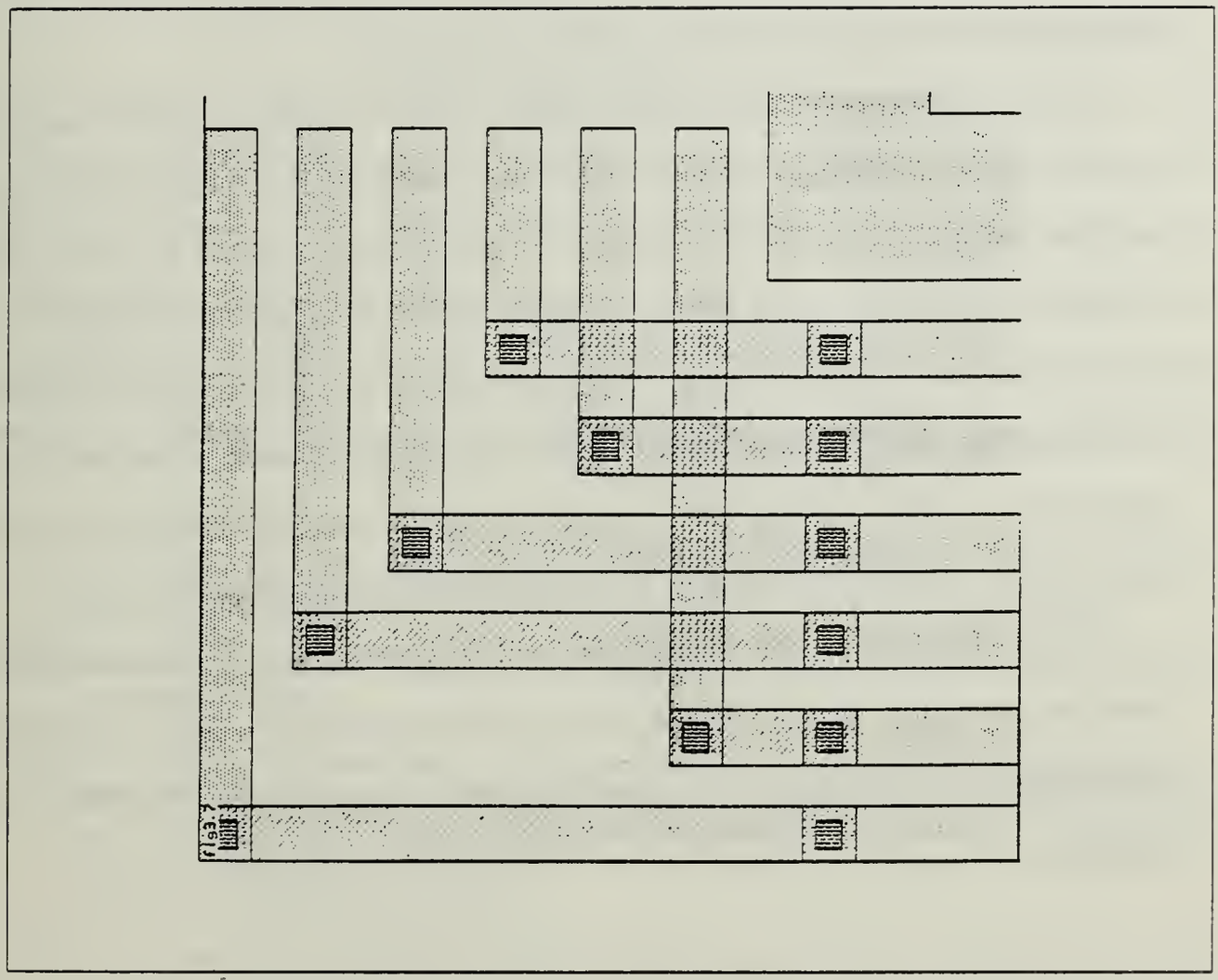

Figure 26: Routing moat corners in MacPitts

D. SUMMARY

MacPitts' routing methodologies are very inefficient. The choice of a one-layer river router to interconnect data-path and controllers is unfortunate because it forces a terminal ordering in the controller that usually results in non-optimal controller designs. Polysilicon is not an adequate layer to employ in the river router because of the long distances involved. Significant transmission delays result even when wiring small circuits (less than 500 transistors).

The pad routing algorithm is also deficient. Pads are evenly divided and placed on the top, right and bottom sides. Pads are never located on the left side. If any one side cannot accomodate its pad quota, it is extended until it can. No attempt is made to determine if the other two sides can accomodate the "overflow". The final placement is determined by an integer assigned to the pad by the designer in the source file. Pad 1 is located on the top-left, pad 2 is located immediately to the right of pad 1, etc. The last pad is located on the bottom-left (see Figure 24). No effort is made to place a pad near its internal connection terminal.

Finally, it is possible to route the pad routing channels with one layer. However, since MacPitts does not coordinate track allocation between the four ring channels, polysilicon and poly-metal cuts are always used to connect net segments at each corner.

IV. ROUTING IN THE MONTEREY SILICON COMPILER

The inefficient circuit designs produced by MacPitts is exemplified by the 251 transistor layout in Figure 27. This chip has dimensions of 1.78 mm and 1.74 mm for a total area of 3.10 mm^2. A circuit performing the same functions can be handcrafted into a much smaller design. Inefficiency can also be measured by performance. Large circuits tend to be slower because of increased routing distances. There are various reasons why MacPitts' designs exhibit poor size and performance characteristics. This investigation has focused on the pad placement and routing algorithms. A detailed description of MacPitts methodology in this area was presented in Chapter III. From Figure 27 the following problems can be observed:

1. All signals from the pads must enter the circuit body through the left side. As a consequence, long wires and wide routing channels are generated.
2. No effort is made to minimize wire length by placing pads near their internal connection points.
3. Even when sufficient space exists around the periphery to accommodate all pads, circuit dimensions may be increased if a side cannot accommodate its pad "quota". The circuit in Figure 27 was extended in both the vertical and horizontal directions. This resulted in the large empty areas to the right and above the internal circuit body.
4. Unnecessary use of polysilicon and vias resulting from an inadequate track allocation algorithm and the poor placement of the ground and power pads. The high resistivity of polysilicon and the high capacitance of vias degrade circuit performance.

This chapter introduces a pad routing strategy tailored to the MacPitts circuit architecture. The algorithms are written in Franz Lisp and are included in Appendix B. As in previous chapters, the LISP functions used in the algorithms will be denoted by boldface fonts. Their arguments are denoted by italics.

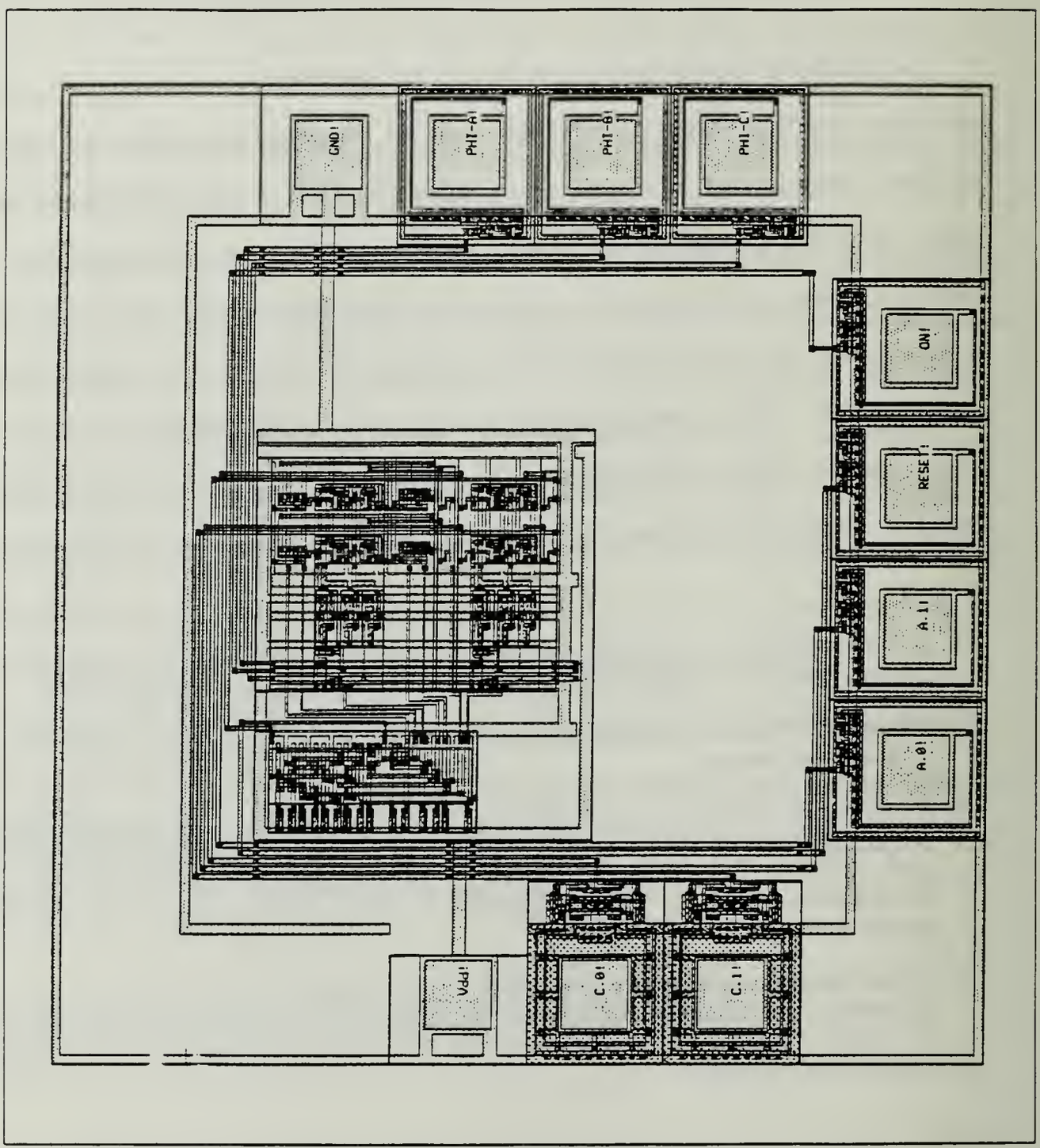

Figure 27: Typical MacPitts circuit

A. DATA-PATH

One of the more conspicuous and peculiar characteristics of circuits designed by MacPitts is that all signals from the pads must connect to the internal circuit through the left side. This behavior has a drastic adverse effect on circuit performance. Depending on the specific circuit geometry, circuit speeds may be halved. The worst possible scenario is depicted in Figure 27, where an output pad on the right side and very close to its internal connection point, must be routed clear across the chip to enter the circuit. From here it is routed, in polysilicon, across the entire data-path. A second ill effect is that channels become progressively more congested and wider than is necessary. The mechanism that results in such routing nightmares was discussed in Chapter III.

The solution to this problem is simple and direct. As discussed in Chapter III, all nets must enter through the left side because the macro **make-left-tip** is used to attach the attribute 'left' to all data-path terminals that need to connect to the pads. Changing the macro to **make-right-tip** makes all such nets connect to the right side. Unfortunately the same relation does not apply with regards to the top and bottom sides, since the bus infrastructure available for routing signals in the horizontal direction, does not exist in the vertical direction. Consequently, it was decided to limit the accesibility of data-path to the right and left sides only.

The ideal solution to the problem is to divide the data-path in half. Anything to the left of center should route to the left side. Conversely, anything to the right of center, should route to the right side. Since the exact length of the data-path is not known during the bus construction phase, this mechanism is not feasible. A useful indicator of distance that is available, however, is the number of units required by data-path. Since units are laid sequentially across the length of data-path, they provide a rough measure of data-path length.

The Monterey Silicon Compiler uses the unit number to determine if terminals within a unit that connect to pads should route either left or right. If the number of the unit being processed is less than half the number of total units, its terminals will route to the left, otherwise they route to the right. This capability was implemented by means of a two case conditional within **get-basic-buses-from-port-output-unit**. The function is included in Appendix B. Redesign of the circuit in Figure 27 with the changes discussed in this section results in the circuit shown in Figure 28. Notice that pads can now enter the data- path through either the left or right sides. Much area is wasted in both circuits. The mechanism which determines the final chip size depends on pad placement and not on the changes introduced by the new version of **get-basic-buses-from-port-output-unit**.

B. PAD ROUTING

There are three parts to the new pad routing process: pad-placement, net extraction and net layout. The procedures exploit changes in the data-path routines that allow signals to enter through either the left or right sides. These changes are discussed in Chapter IV.A. Pads may be placed on two, three or all four sides. The number of sides ultimately used is determined by a chip area optimizing algorithm. Every effort is made to accommodate all pads into the space available. If the pads require more space than is available, the longest side of the chip is extended until all pads fit. By extending the longest side, the total increase in area is minimized. Finally, polysilicon and via usage has been drastically reduced by proper placement of the ground and power pads and by an efficient track allocation algorithm.

The pad routing functions are in the file frame.l. They are invoked by a call to **layout-pins** in **layout-object**. **Layout-pins** produces the layout information that results in the pad ring shown in Figure 29. As in MacPitts, the final version of *pins-layout* is obtained by running **layout-pins** twice. This is necessary because the pad routing channel width requirements cannot be calculated until some idea of

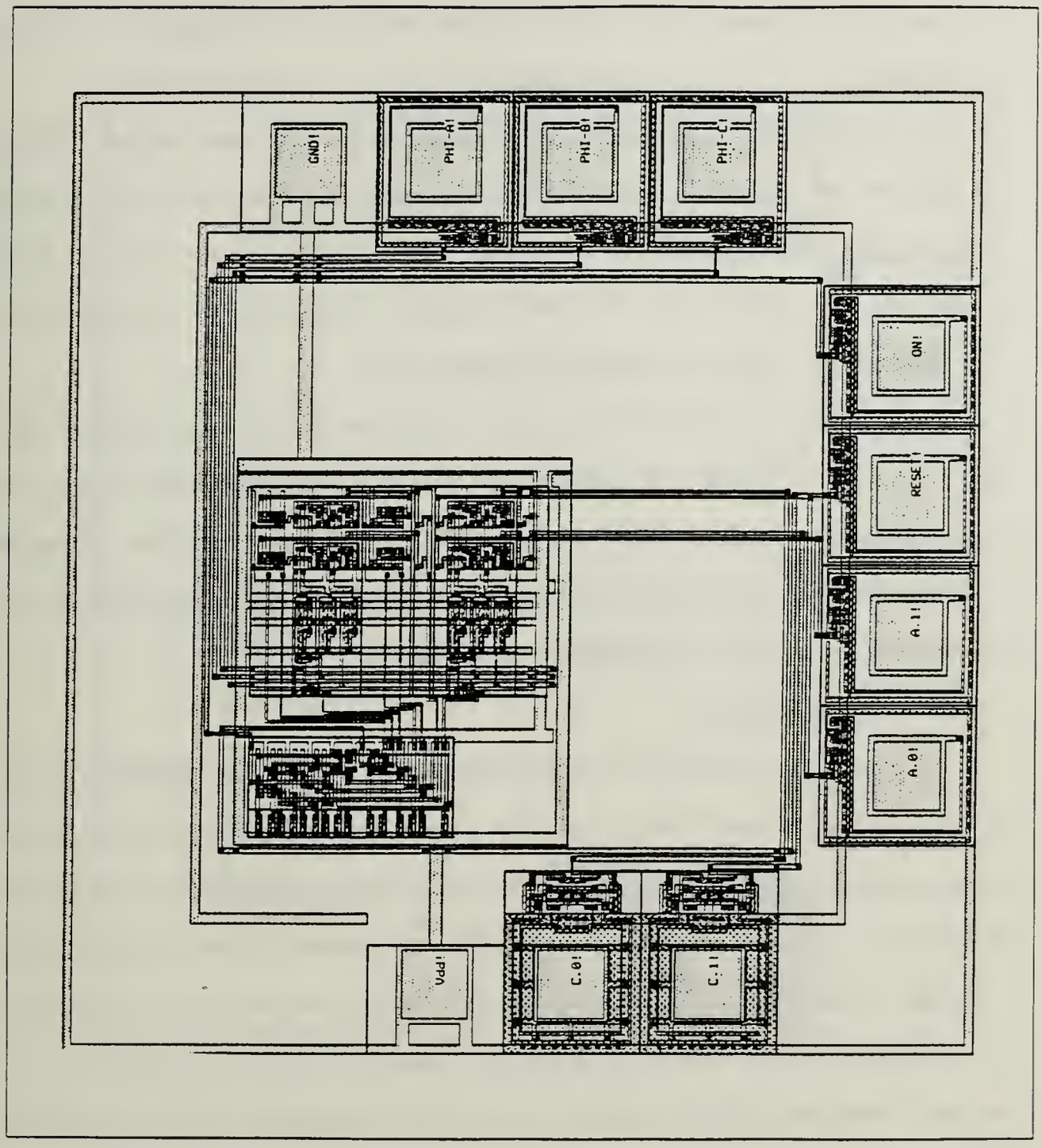

Figure 28: Opening data-path on the left and right sides.

the pad positions is available. During the first run through **layout-pins**, widths of 0 are used. The results from the first run are used by **get-ring-width** to calculate exact width requirements for the routing channels. These four values, representing the channels widths of the top, right, bottom and left routing regions, are stored in *ring-width*.

Once *pins-layout* is produced, the net-extraction processes begins. The algorithm uses net information contained in *internal-layout* and *pins-layout* to build two net-lists. The first list, *left-ring-nets* includes all nets that enter the internal circuit through the left side. The second net-list, *right-ring-nets* corresponds to all nets that enter the circuit through the right side.

The final phase performs the actual routing process. It too is divided into left and right sides. The left side uses *left-ring-nets* as an argument, while the right side uses *right-ring-nets*. Each side is further divided into three problems: routing from pads on the bottom, routing from pads on the side, and routing from pads on top. The routing process is initiated by a call to **moat**.

1. Pad Placement

The new pad placement strategy differs from the original MacPitts' methods in two significant ways. First, it finds the order of net terminals in *top-part* and *wing-span* and builds a pad terminal list in the same order. Not only does this result in a reduction of the average distance between net terminals, it also simplifies the final routing. As discussed in Chapter 3, MacPitts builds the pad lists directly from the source file. It has no built-in optimization capability. Second, pads may be placed on two, three or four sides. The criterion for this decision is chip area reduction. In contrast, designs by MacPitts always use three sides for pad placement.

The pad placement process is initiated by a call to **arrange-pins** from **layout-pins**. **Arrange-pins** is responsible for selecting the sides that pads are to be placed in. It does this by means of a three case conditional. The first option

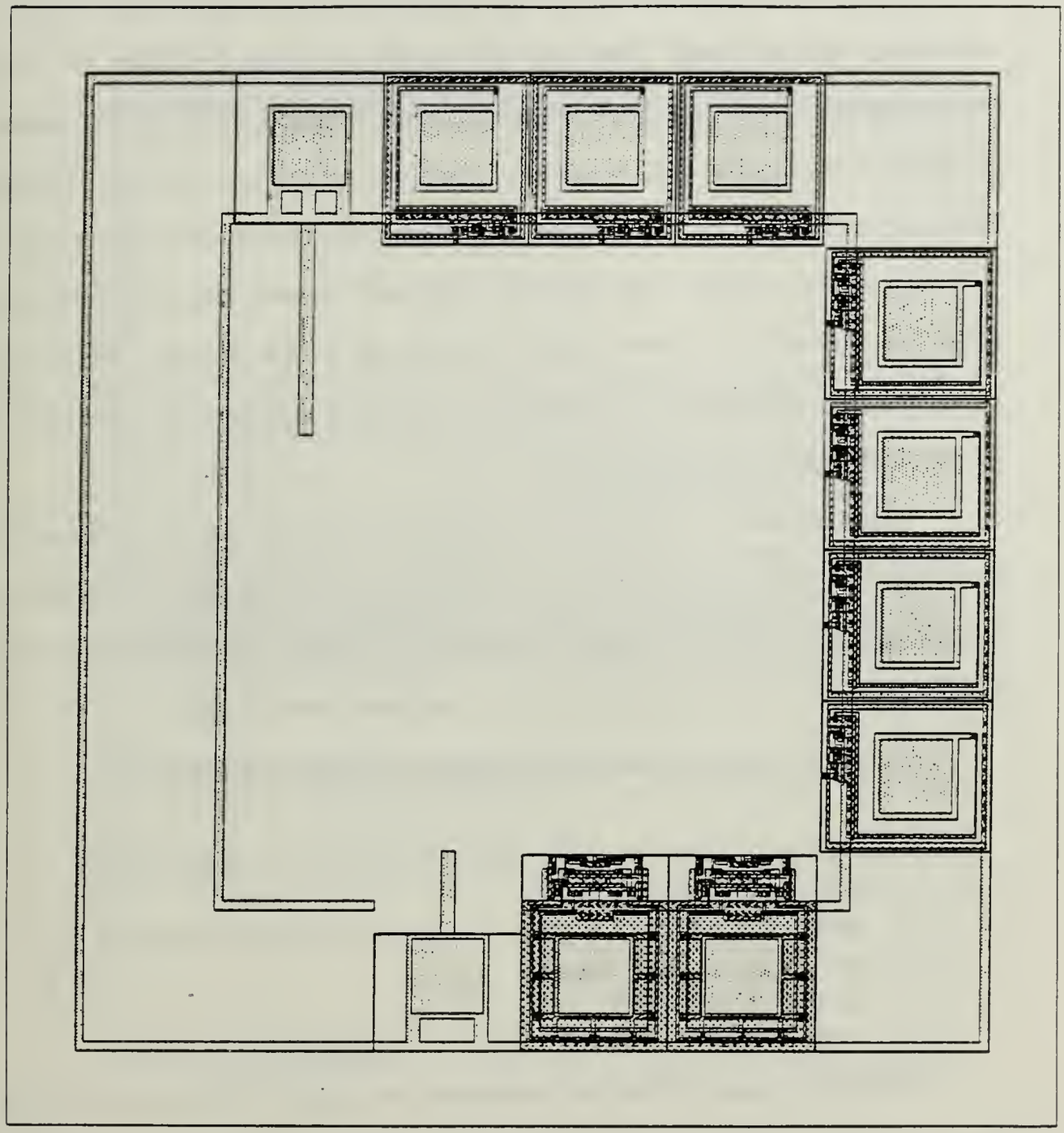

Figure 29: MacPitts' pad ring.

considers if all pins can fit along the top and bottom sides. If not, the possibility of using top, bottom and either the right or left side is tested. If this strategy is inadequate, all four sides are used.

Notice that, regardless of the placement approach, the top and bottom sides will always contain pads. There are two reasons why these two sides are "common denominators" in the pad placement schemes. First, as Figure 30 illustrates, the skeleton ground rail is only exposed to pads along the top boundary. This configuration provides strong justification for placing the ground pad on the top. Since it is advisable to maintain ground and power pads distant from each other to avoid latch up problems, the power pad is best placed in the bottom. Once a side is extended to provide space for one pad, placing more pads on that same side results in no additional area requirements.

A second reason for having the top and bottom sides as common denominators is that the top and bottom sides are usually longer than the right and bottom sides. Circuit growth in the horizontal direction is a function of processing complexity. Vertical growth, on the other hand, is a function of word length.

Arrange-pins yields a list named *pin-net* of the following form:

```
(4 ((((1 (signal input (reset))) (2 (on)) (3 (phia)) (4 (phib))) left)
(((1 (phic)) (2 (ground))) top)
(((1 (port input (a 1))) (2 (port output (c 1))) (3 (port input (a 0)))
(4 (port output (c 0)))) right)
(((1 (power))) bottom)))
```

The first element of the list indicates the number of sides selected for pad placement and can be 2, 3, or 4. It must be followed by the same number of lists. Each list contains a list of pads and terminates with the side where the pads in that list are to be placed. Each pad is assigned a number. The product of this number and the pad width provides the pad position (x-coordinate when placing pads on top or bottom, y-coordinate otherwise). In this example, the leading 4 indicates

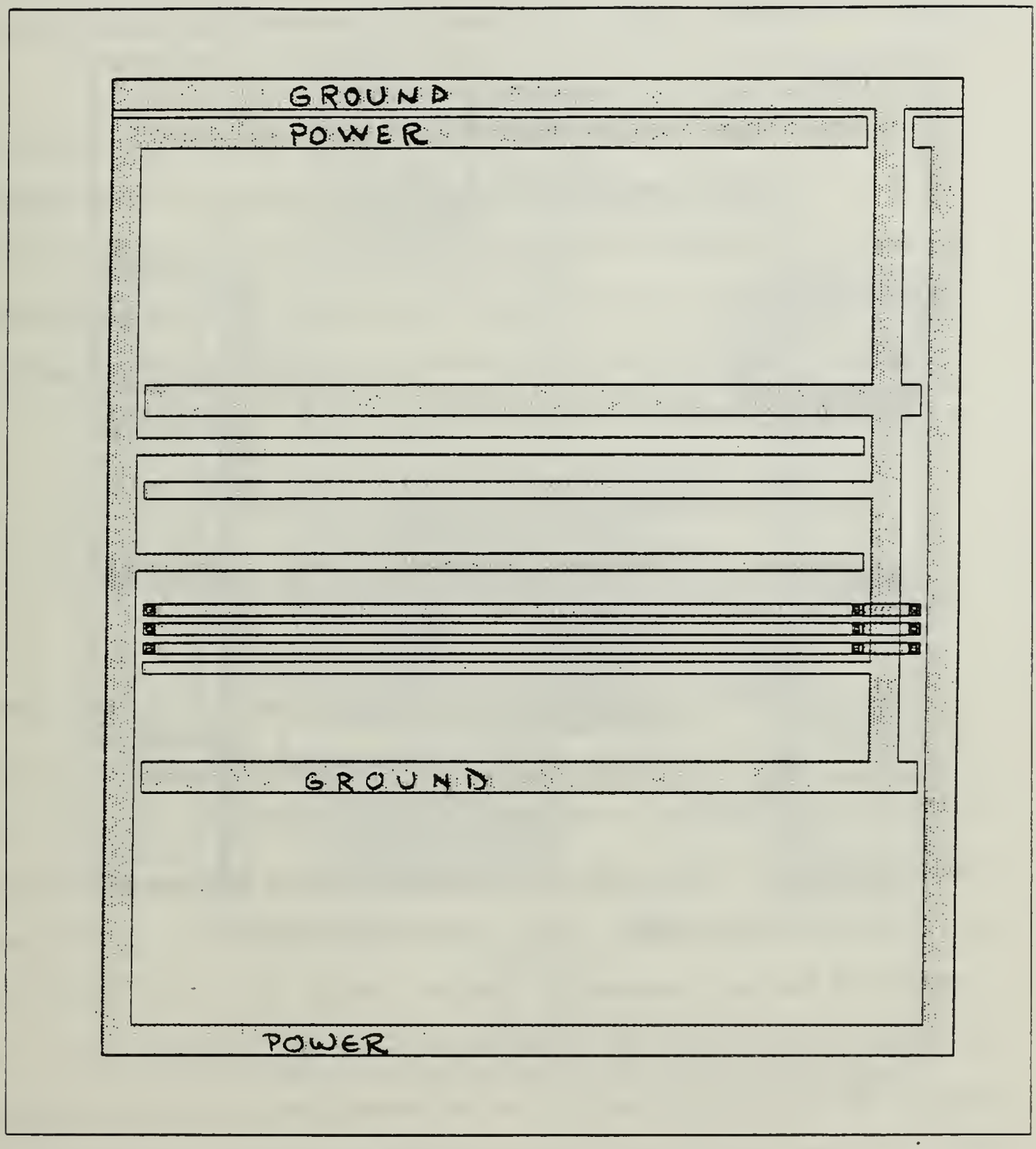

Figure 30: MacPitts' power and ground frame

that the enclosed pad list consists of four lists. Each of these lists contains a list of pads and a side attribute. This attribute can take either top, left, bottom or right as a value. As their names suggest, attributes indicate the side where that list of pads is to be placed. Figure 31 illustrates the placement that results from the list in this example.

Arrange-pins operates on *pins, sorted-pins, extended-right* and *extended-top. Pins* is extracted from the circuit source file. It consists of a pad name and a pad number. Pad numbers are assigned by the user in the source file. *Pins* is the only list that MacPitts uses to construct the pad layout. It provides no information with which to make intelligent placement decisions. The *pins* list that corresponds to the previous example may take the following form:

(((ground) 1) ((phia) 2) ((phib) 3) ((phic) 4) ((on) 5) ((signal input (reset)) 6)
((port input (a 0)) 7) ((port output (a 0)) 8) ((port input (a 1)) 9) ((port output (c 1)) 10) ((power) 11))

The list *Sorted-pins* provides information that results in more effective pad placement. The list is created by **extract-internal-nets** operating on *top-part* and *wing-layout. Top-part* contains the exact position of those terminals in data-path that connect to pads. *Wing-layout* contains similar information with respect to terminals in the controller. *Sorted-pins* is made up of two lists. The first list identifies all internal terminals that connect through the left side. This includes all points in *top-part* with the attributes 'ring' and 'left', and every point in *wing-layout.* The second list contains all *top-part* points that have attributes 'ring' and 'right'. The attribute 'ring' is used to identify all points that participate in the pad routing process. The 'left' attribute indicates that the point is located on the left edge of data-path. The 'right' attribute indicates that the point is located on the right edge of data-path. These two lists are then sorted from the smallest to

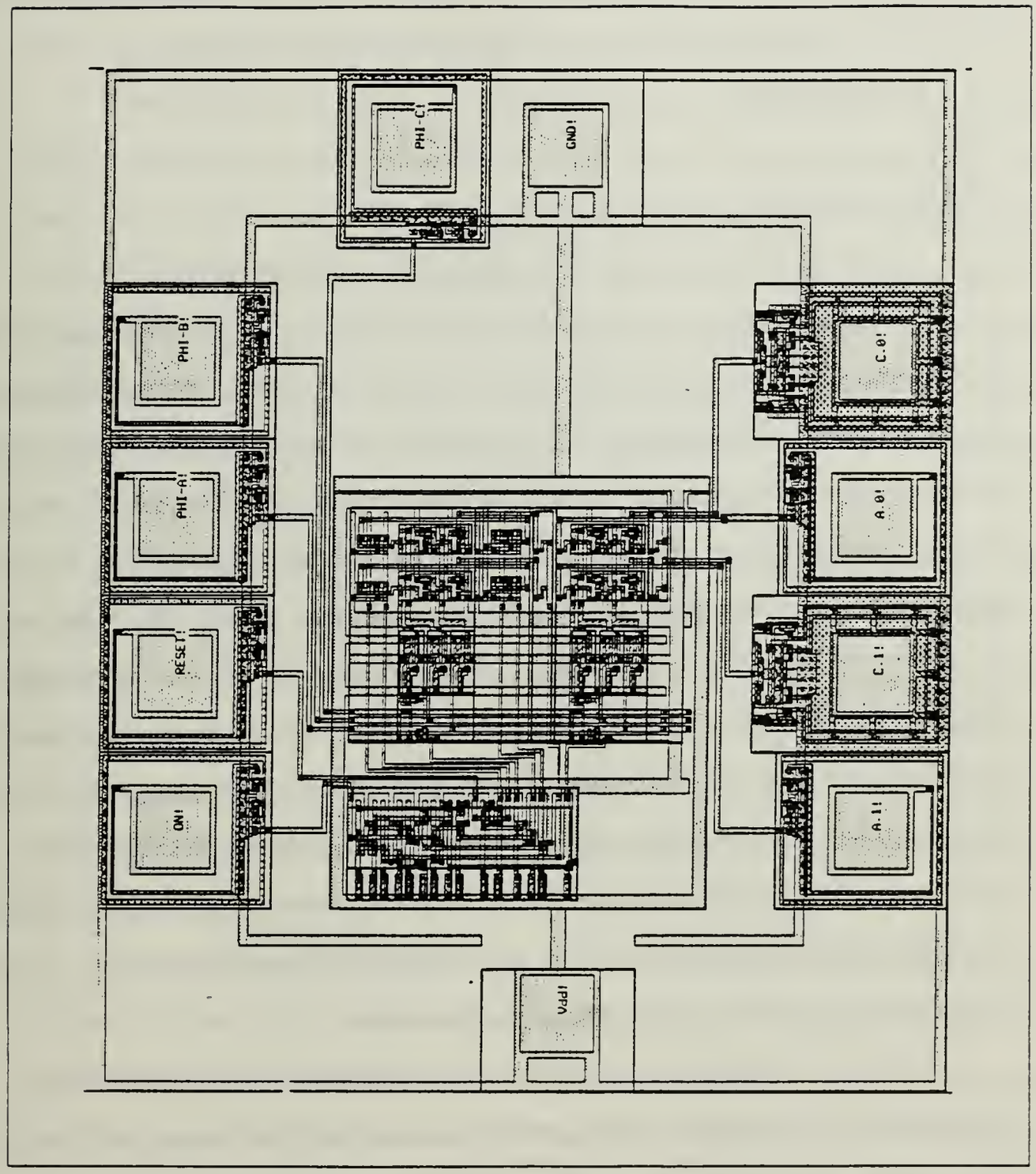

Figure 31: Placing pads on four sides.

the greatest y-coordinate. This service is provided by **sort-by-y**. **Arrange-pins** dismantles the list *sorted-pins* into two lists. The first list is assigned the name *left*. The second is given the name *right*. The *sorted-pins* list corresponding to the left side of the example above is:

(((reset) (on))

The list corresponding to the right side is:

((input (a 1)) (output (c 1)) (input (a 0)) (output (c 0))))

As the example shows, *sorted-pins* and *pins* are two very different lists. First, *sorted-pins* is missing a number of pads. In fact, ground, power, phia, phib and phic are never present in *sorted-pins*. These pads are treated differently because, when routing is concerned, they do not interact with the internal circuit as other pads do. Ground and power connect to the skeleton, not the circuit body. The clock signals: phia, phib and phic, are unique in that, for any circuit, they may connect to either the right, and/or left sides. This provides some latitude in making the final pad lists. When on the left side, clock pads are always located between the *wing-layout* and *top-part* terminals. On the right side, they are always the bottom three terminals. Of the three, phia is always on the bottom, phib in the center and phic on top. Clock pads are appended one at a time to the shorter of the two lists in *sorted-pins*. Since the list lengths are revaluated after each insertion, all three pads need not always be connected to the same side.

Pins and *sorted-pins* also differ in the structure of their list elements. A net referred to as (input (a 0)) in *sorted-pins* is called (port input (a 0)) in *pins*. The difference is that the elements in *sorted-pins* are net names. The elements in *pins* are pin names. Pin names not only identify signals names, they also identify the type of pad. In the example, the net name is (input (a 0)). The pad type is port input. Some other pad types include: tri-state, port output and i/o4. This

difference drives the requirement for *pins* in **arrange-pins**. Once the pin layout lists are formed in **arrange-pins**, **order-pins** is called to transform each partial pin name into the complete pad name. **Order-pins** finds each element of the layout list in *pins* and imports the missing name parts.

When all pads can fit into the top and bottom sides **pad-on-two-sides** is called. This function uses four arguments: *left*, *right*, a list consisting of the clock pads, and the difference in the number of elements between *left* and *right*, to build a pad list that results in the placement of all pads on the top and bottom sides. It attempts to balance the number of pads in *left* and *right* by transferring clock pads to the shortest of the two. Finally, it appends the power pad to *left*, and appends ground to the *right*. *Left* is placed along the bottom, and *right* is placed along the top. Figure 32 illustrates the pad layout for this configuration.

When three sides are required to accommodate all pads **pad-on-three-sides** is called. This function uses *left*, *right* and the clock pads to build a pad list that results in the placement of all pads on the top, bottom, and either the left or right sides. The clock pads are appended to the shorter list between *left* and *right*. When there are more pads in *right* than in *left*, the pads in *left* are placed along the bottom. Any remaining pads are placed on top. Pads in *right* are placed along the top and right sides. Any remaining pads in *right* are then placed along the bottom.

Placement of pads on all four sides is directed by **pads-on-four-sides**. As in all previous cases, clock pads are affixed to the shorter list between *left* and *right*, power is the first pad in *left*, and ground is the first pad in *right*. The algorithm starts with the left list. It assumes that the best solution is one that ensures that the left side is completely used. It does this by finding the difference between the number of pads that fit on the left side and the number of pads in *left*. If the difference is one or less, it positions power on the left corner of the bottom side, and all others on the left side. Otherwise, it divides the difference by two, and places

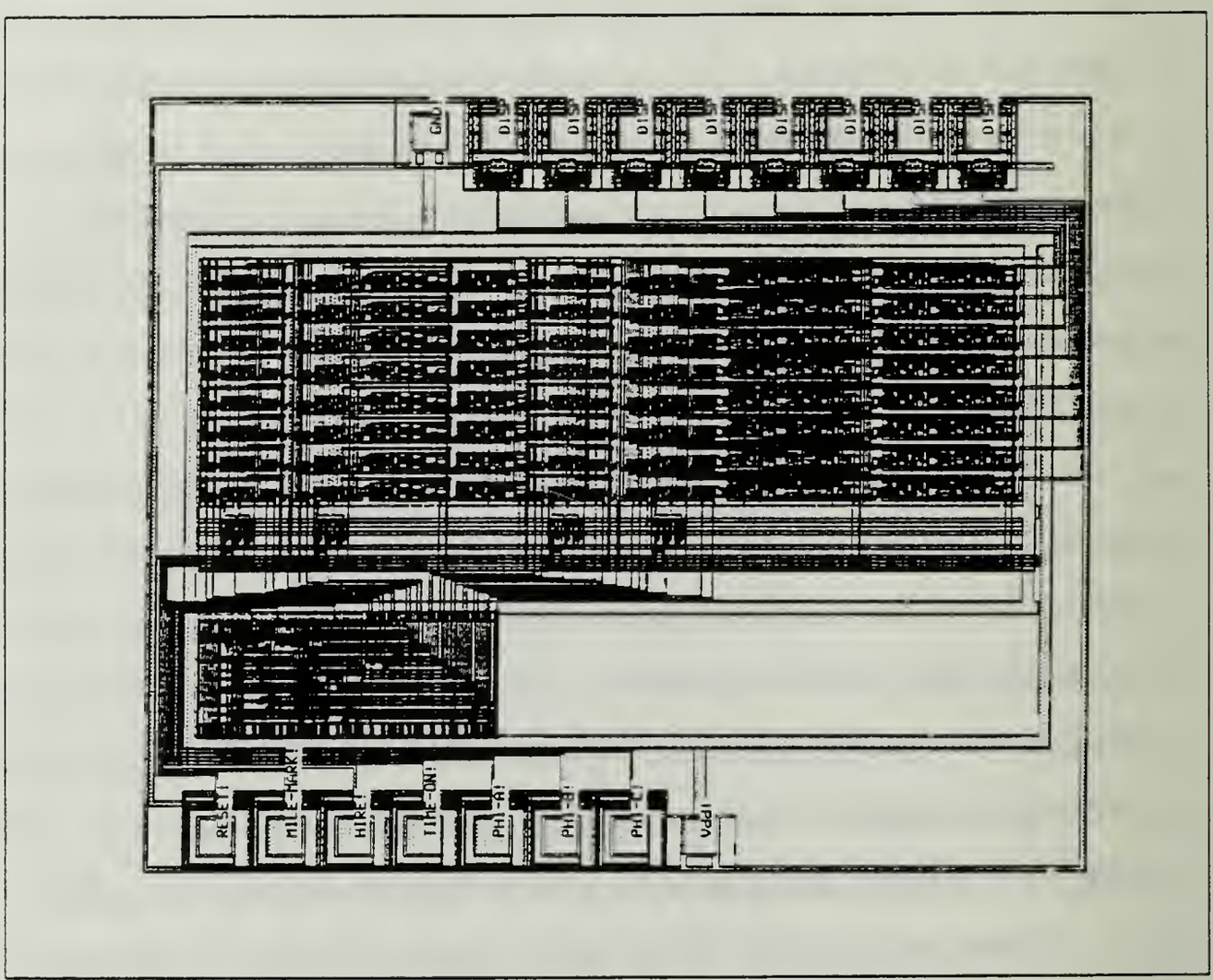

Figure 32: A circuit with pads on two sides.

this number of pads on the bottom, fills up the left side, and places the remainder on top. Pads in *right* are placed in a similar fashion. If the difference between the number of pads that fit on the right side and the number of pads in *right* is one or less, the ground pad is placed at the right corner of the top side. Otherwise, the difference is divided by two, and that number of pads, starting with ground, are placed on top, then the right side is filled, and finally, any excess is placed along the bottom. Figure 31 illustrates a circuit with pads on all four sides.

In all pad placement schemes, ground is the first pad placed when routing *right*. It is always located on top. It serves as a boundary; pads to the left of ground connect to the left side of the circuit, while those to the right of ground attach to the right side of the circuit. In a similar fashion, power is the first pad placed when routing *left*. It is always on the bottom. It also serves as a boundary. To its right, all pads attach to the right side. To its left, pads connect to the left side of the circuit. This characteristic of the ground and power pads is exploited during net extraction. Placing ground and power pads in this fashion eliminates all occurrences of nets crossing over the power/ground pad to skeleton connecting wire. It reduces the number of vias and the amount of polysilicon required.

2. Pad Layout

Once *pin-net* is formed, pad layout is initiated by **place-pins**. The list produced by **place-pins** is named *pins-layout*. This list contains the information required to produce pad layouts such as the one illustrated in Figure 33. **Place-pins** peels one list from *pin-net* at a time and transfers it to **place-pins1**. **Place-pins1** extracts the side where pads in that list are placed, the pad name and the pad number and forwards it, one pad at a time to **place-pin**. **Place-pin** uses the pad name, pad number and side information to determine the type of pad, its exact location, and its orientation.

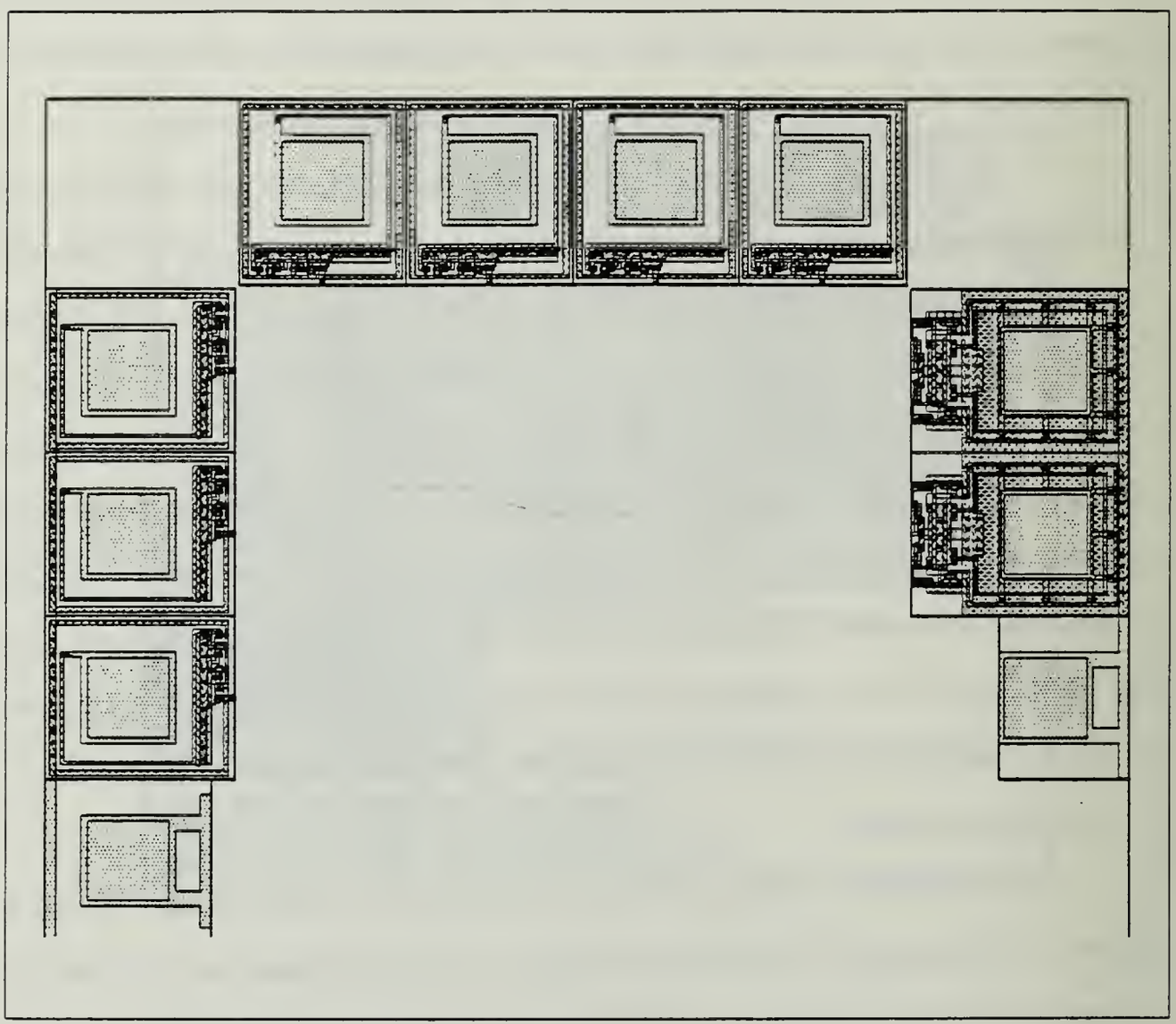

Figure 33: Pins-layout

Place-pin calls on **layout-pad** to identify the pad type from among the following list:

power	ground	phia
phib	phic	output4
output8	input	tri-state4
tri-state8	i/o4	i/o8

Once the pad type is determined, **layout-pad** calls on the appropriate function that returns a list specifying the actual pad layout. These functions are defined in the file pads.l. The following call results in the description of the pad in Figure 34:

(**layout-pad20b-input-pad** *power* (**input-pad- name** *pad*)
(**input-pad-in-wire** *pad*) *side*)

The function **layout-pad20b-input-pad** specifies the type of pad. In this case an input pad of type pad20b. A second type of pad class is rinout. A call to this type takes the form **layout-rinout-input-pad**.

Layout-pad calls on the **input-pad-name** to label the pad with the name specified by the parameter *pad*, and **input-pad-in-wire** to label the point on the pad where the internal circuit must connect to. The parameter *power* is the width required by the pad-layout ground and power ring.

To position the pad within *pins-layout*, **place-pin** uses provides the pin number and side to the L5 function **move**. **Move** takes three arguments: item, x-distance and y-distance. To place a pin on the right side, **place-pin** calls:

(**move** (**rotccw** (**layout-pad** *pin power 'right))*
right
(* (**pad-class-width**) (**1-** *pin-number*)))

In this example the pad to be moved is brought in by **layout-pad** and rotated counterclockwise by the L5 function **rotccw**. The pad is moved in the x-direction by the parameter *right*. This parameter is obtained by the function **pins-dimensions** and is the sum of *extended-right*, the width of the right ring channel and the span

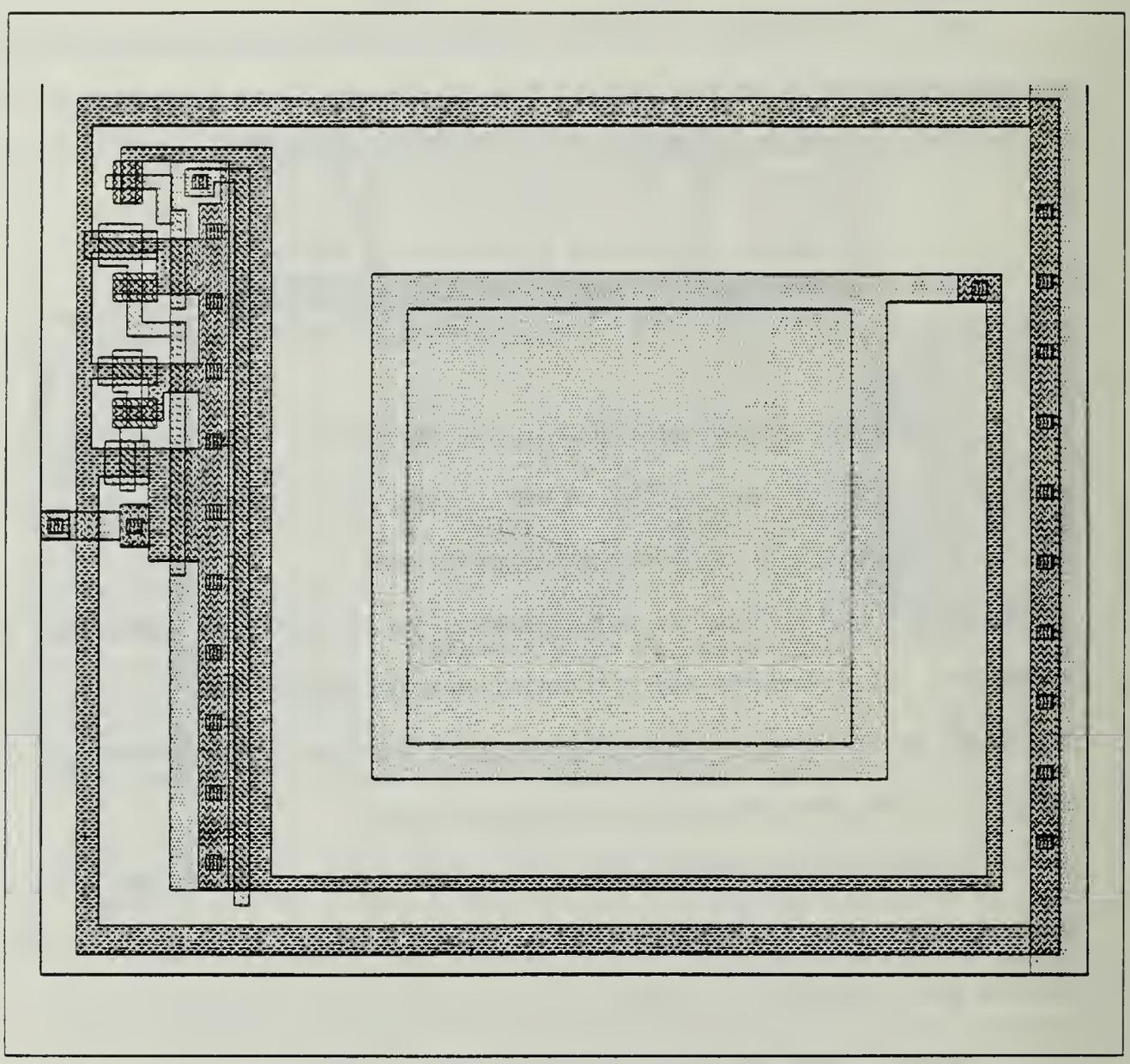

Figure 34: Results from **layout-pad20b-input-pad**

across the large power and ground rails that surround the internal layout. The pad is moved in the y-direction by the product of **pad-class-width** and pin-number−1. **Pad-class-width** has no arguments. It returns the width that corresponds to the pad type in question. For the pad20b class, this parameter is 128λ. It is 100λ for the rinout class.

3. Net Extraction

The purpose of the net extraction process is to build net-lists. A net-list specifies all terminal locations of a net. In the pad routing process, net extraction must identify net terminal locations in *internal-layout* and *pins-layout.*

The net extraction functions in the Monterey Silicon Compiler exploit design characteristics, inherent to the Monterey silicon compiler, to simplify the algorithms. First, since the pads are ordered in the order that nets appear inside the circuit, once the first is found on both lists, the other nets fall out automatically. Second, pads that connect to the left side of the circuit are always left of the ground and power pads. Pads that connect to the right side of the circuit are always to the right of the power and ground pads.

The net extraction process described here performs well when confronted with two terminal nets. It is not equipped to handle nets with three or more terminals. Multi-terminal nets occur when more than one tri-state pad is used in a circuit. The control signal used to change the impedance state of tri-state pads usually form a multi-terminal net. The Monterey pad router does not currently have this capability.

Two net-lists are formed during net extraction. The first, given the name *left-ring-nets*, consists of nets that connect to the left side of the circuit. *Right-ring-nets* includes all nets that connect to the right side of the circuit. Since they are formed by almost identical processes, only *left-ring-nets* is discussed here.

Left-ring-nets is a list of two lists. The first list contains the y-coordinates of all the internal terminal points. This list is obtained by means of the functions **sort-y** and **get-left-nets1** in much the same way that was described in the extraction process that led to pad placement. Essentially, all those points in *internal-layout* with both the 'left' and 'ring' attributes are extracted, then sorted by increasing order of their y-coordinates.

The second list contains coordinates for terminal points at the pads. The values in this list are interpreted as follows:

1. If $value < 0$, then the point is on the bottom side, and $abs(value)$ = x-coordinate.
2. If $value >$ *extended-top*, then the point is on the top side, and $value -$ *extended-top* = x-coordinate.
3. If $0 < value <$ *extended-top*, then the point is on either the left or right side and $value$ = y-coordinate.

These values are obtained by applying **prep-pad-bank** to *pins-layout*. Since the order of pads in *pins-layout* is derived from the order of terminals in *internal-layout*, the first element in the first list connects to the first element in the second, the second element in the first list corresponds to the second element of the second list, and so on. A circuit with *extended-top* = 100, and pads on the bottom, left and top that connect to the left side, could have the following *left-pad-bank*:

((10 20 30 50 70) (-40 20 40 70 130))

Prep-pad-bank serves three functions. First, it finds the x-coordinates for both the ground and power pads. Second, it divides *pins-layout* into two lists. Pads to the left of the power and ground pads are packaged in a list and passed to **left-pad-bank**. Pads to the right of the power and ground pads are passed on to **right-pad-bank**. Both functions represent each pad into a single number. This number is sufficient to locate the pad terminal. The terminal numbers are assigned as follows:

1. If the pad is placed on the right or left side, net_{pad_i} =y-coordinate$_{pad_i}$.
2. If the pad is placed on top, net_{pad_i} =*extended-top*+x-coordinate$_{pad_i}$.
3. If the pad is placed on the bottom, net_{pad_i} = 0−x-coordinate$_{pad_i}$.

4. Net Layout

The goal of the net layout process is the interconnection of the net-lists constructed during net extraction. Of course, this task must be performed within the framework of the design rules. The proposed net layout method is customized to the Monterey Silicon Compiler target architecture. It combines routing strategies found in the moat and river router. These routers were discussed in Chapter II.B.4 and II.B.5. The Monterey Silicon Compiler pad router is designed to minimize polysilicon and via utilization in the routing channels. It does not guarantee the smallest possible channel width. The pad router is a river router with a river bed in the shape of a rectangular ring. The routing area is made up of four rectangular channels and the four square areas at the corners. The rectangular areas are located between the outer skeleton ground and power rails, and the interior boundary of the pad ground ring. Figure 35 illustrates the pad routing area.

The differentiation between nets that connect to the left side of the circuit, and nets that connect to the right side is also observed by the net layout functions. The pad routing process is initiated by a call to **moat**. **Moat** has three arguments: *left-ring-nets, right-ring-nets* and *ring-width*. *Ring-width* is a list of four numbers. These numbers represent the ring channel widths for the top, right, bottom and left sides. The pad router functions use the macros **top-width, right-width, bottom-width** and **left-width** to access the desired value from *ring-width*. **Moat** has only one function, to direct the appropriate net-list to **route-left-bottom** and **route-right-bottom**. Since **route-left-bottom** and **route-right-bottom** perform their functions in the same way, this discussion will be limited to the left instance only.

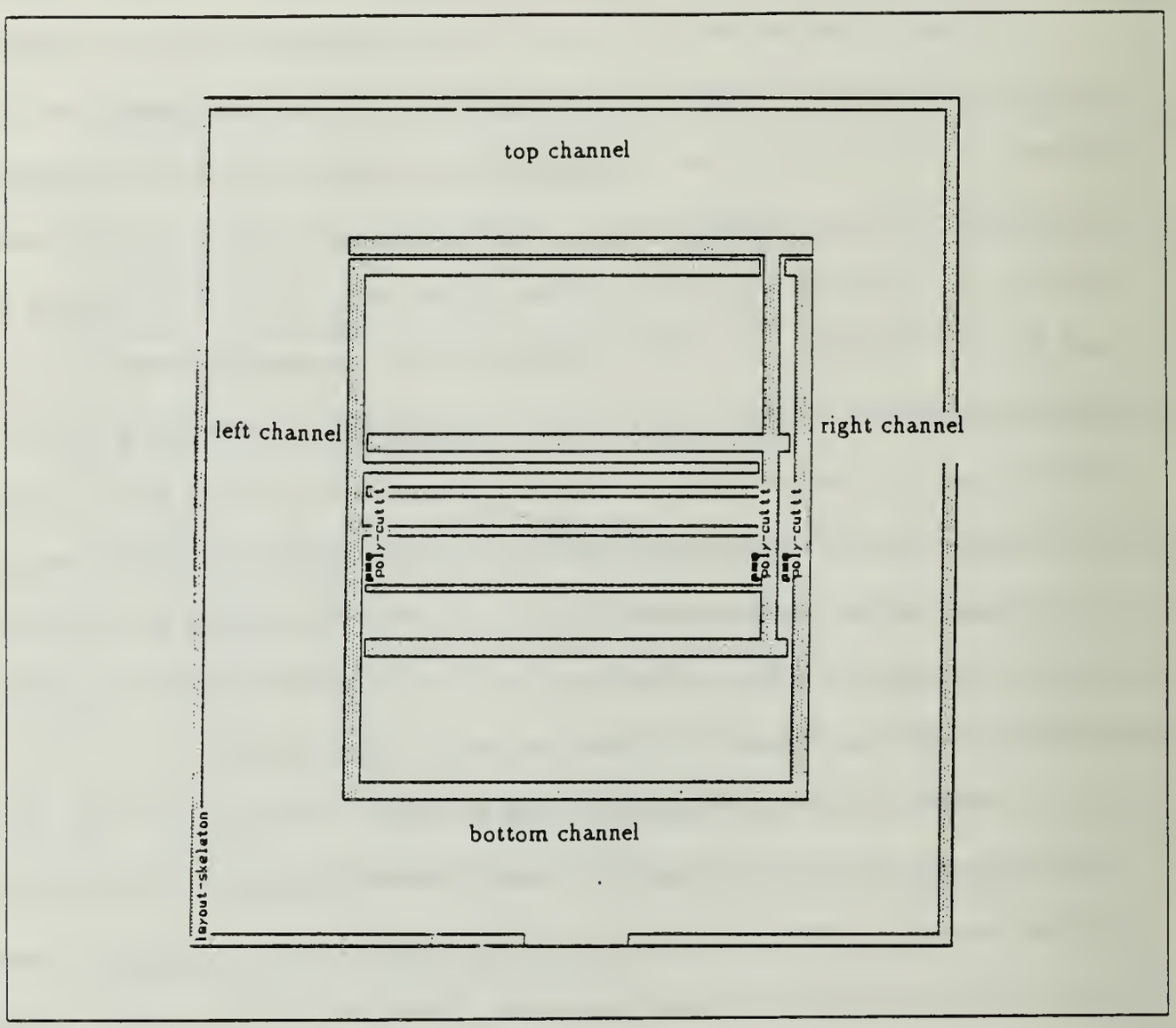

Figure 35: Pad routing area

The most complex type of pad routing problem that the Monterey compiler can be presented is shown in Figure 36.a. The routing area for this instance of *left-ring-nets* consists of the left portions of the bottom and top rectangular channels, the entire left channel, and the corner areas at each end of the left channel. The important parameters in this routing problem are the net-lists and the edges of the top and bottom boundaries of the left routing channel occurring at $y = 0$ and y =extended-top. The net-list is important because it identifies the terminals that need to be routed. The edges of the left channel are important because they identify the ring corners.

During the discussion of routers in Chapter II, it was determined that the moat router is nothing more than a modified channel router. Since the net-lists are sorted, i.e., the ith element of the terminals on the left side belongs to the same net as the ith element of the terminals on the right side, this routing problem can be solved with river routing techniques. The only issues to resolve are the track allocation problem and routing around the ring corners. These problems are solved simultaneously by the layout routines.

Tracks must be allocated not only to minimize the occurrence of vias and polysilicon but to prevent shorts caused by overlapping different nets. The track allocation method used is essentially that used by the MacPitt's river router and discussed in Chapter III.B. A differentiation is made between nets that are routed up-river (the y-coordinate of the internal terminal is greater than the y-coordinate of the pad terminal), nets routed down-river (the y-coordinate of the internal terminal is less than the y-coordinate of the pad terminal), and nets routed across river (y-coordinate of both terminals are equal). Picturing the routing area as divided into a number of concentric tracks, for each consecutive net routed up-river the assigned track is incremented by one. That is, the first net routed up- river is assigned to track 1, the second to track 2, and so on. When either a down-river or across river

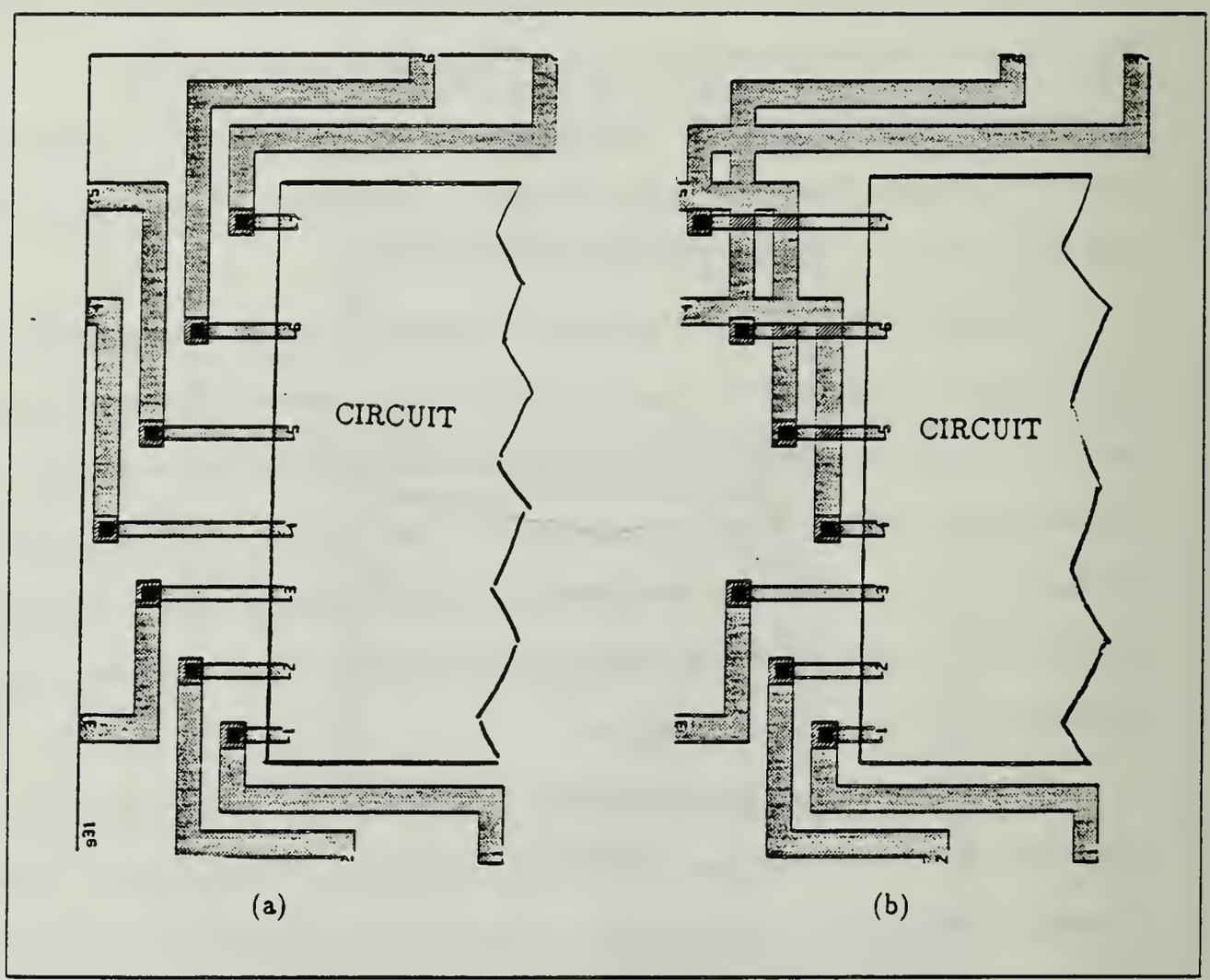

Figure 36: Sample routing problem: (a) routing by Monterey silicon compiler, (b) errors caused by change in routing direction

situation occurs, the track counter is reset to one. The same method is used to assign tracks to consecutive nets routed down-river. In this instance, however, it is an up-river or across river net that resets the track counter to one. Every time an across river situation occurs the track counter is reset. Once a net is assigned a channel it remains in that channel until it reaches its destination.

Tracks must be allocated to prevent interference when routing subsequent nets. The algorithm routes the lowest net first, using **route-left-bottom**. After all nets with pad terminals on the bottom are routed, what remains of *left-ring-nets* is passed to **route-left-side**. **Route-left-bottom** recognizes pad terminals from the bottom side because their net values are less than zero. Finally, if any nets exist with net coordinates greater than *extended-top*, they are passed to **route-left-top** for routing.

If the track allocation method were used as described above, all consecutive down-river nets would interfere with each other. This is illustrated in Figure 36.b. This problem is avoided by reversing the track number order between up-river and down-river nets. For up-river nets track assignment starts with the innermost net and grows outward. For down-river nets track assignment originates with the outermost track and proceeds inward. With this modification, the track allocation method described in the previous paragraph yields the channel of Figure 36.a.

The track that a net is to occupy is determined by the parameter *track*. When routing nets with pads on the bottom, **route-left-bottom** increments this parameter by one for each net it routes. This simplification is possible because pads on the bottom always route up-river. In a similar fashion, nets routed by **route-left-top** always route down-river.

When routing nets with pads on the sides any of the three routing directions may occur. The direction in which the last net was routed is preserved by means of *flag*. This parameter may take one of three values: 'down', 'up', or 'straight'.

Route-left-side determines whether to update or reset the track counter by observing the state of *flag*, and by the routing direction of the current net being routed. The possible outcomes are:

- If current net is straight THEN $track = 1$
- If current net is down-river THEN
  - If $flag = straight$ or $flag = up$ THEN $track = 1$
  - If $flag = down$ THEN $track = track + 1$
- If current net is up-river THEN
  - If $flag = straight$ or $flag = down$ THEN $track = 1$
  - If $flag = up$ THEN $track = track + 1$

Routing around the corners is very simple. The algorithms treat corners as simple extensions of the regular rectangular routing areas. As Figure 37 shows, the net must penetrate into the corner until it reaches the square's diagonal that best aligns with a radial extending from the center of the circuit. In the corner shown in Figure 37, the left bottom corner is simulated by setting the reference point $(0, 0)$ at the top-right corner of the square. The values for both *x-min* and *y-min* are both the negative of the product between the track occupied by the net and the track width. The track width is the sum of the technology dependent parameters *space* and *width*. The track number is the parameter *track*. The *x-max* and *y-max* parameters needed to establish the wire dimensions can be obtained from *x-min*, *y-min*, *width* and the net terminal coordinates.

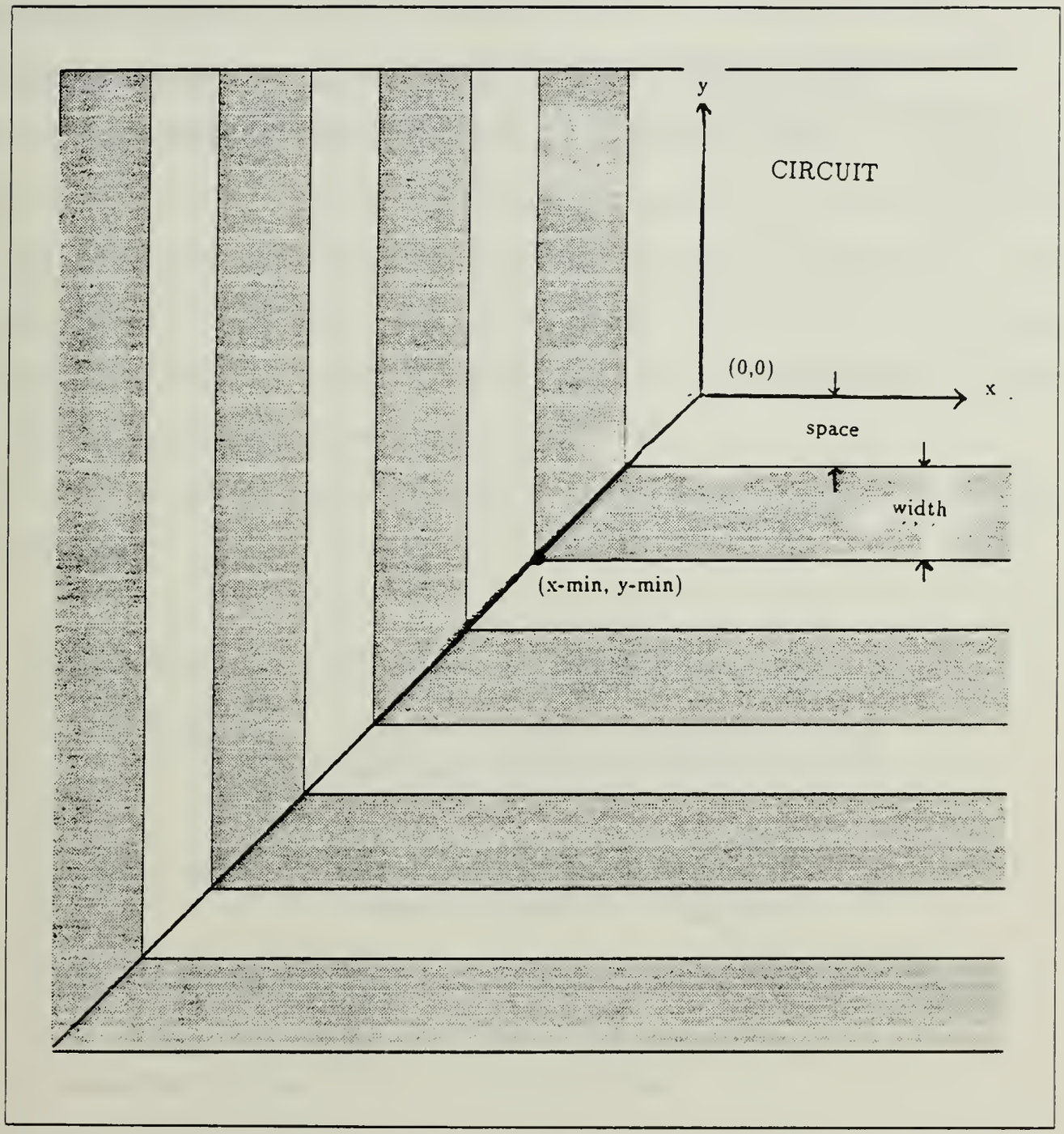

Figure 37: Routing the corners

B. SUMMARY

The pad router developed for the Monterey Silicon Compiler outperforms the original MacPitts pad router in all respects. It requires less area, reduces wire lengths and minimizes the usage of polysilicon and vias. These across-the-board improvements are possible due drastic changes in the pad placement algorithms, and the new capability to enter the circuit body through either the left or right sides.

V. RESULTS

A. MONTEREY SILICON COMPILER ENVIRONMENT

The functions discussed in Chapter IV and included in Appendix B were inserted in the Monterey ISI workstation environment developed by J. Harmon [Ref. 19]. This environment allows MacPitts' circuits to be viewed by MAGIC [Ref. 12]. Changes made by J. Harmon to organelles defined in data-path.l and organelles.l, permit error free extraction of files for simulation. The errors were caused by design rule incompatibilities between MacPitts and MAGIC.

To design environment is created by issuing the following commands on an ISI workstation:

Macpitts	loads MacPitts environment
include *patches*	replaces MacPitts' data-path.l, library and organelles.l with versions created by J. Harmon [Ref. 12]
include *buses*	adds functions that allow connections to the right and left side of data-path
include *pad-router*	adds pad placement, net extraction and pad routing functions
macpitts *file-name*	begins circuit compilation

B. RESULTS

The new pad router was tested on four circuits. Both the MacPitts and Monterey version layouts are illustrated in Figures 38 – 45. Table 1 lists measurements for a number of significant parameters. This particular set of circuits was selected because of their wide range of sizes and internal structure. The source code used

to compile them is included in Appendix C. The first design, MEMORY, is a 2-bit latch with over 200 transistors. It was used extensively during the software development phase because of its small and simple design. TEST is a 3-bit incrementor with over 500 transistors. MULTIP4 is a 1200 transistor 4-bit multiplier designed by D. Carlson [Ref. 3]. Finally, TAXI was presented by Siskind, Southard and Crouch [Ref. 2] in the original MacPitts paper. It is a 1500 transistor, 8-bit taxi meter.

Both MacPitts and Monterey silicon compiler designs were produced for each circuit. Testing consisted of ESIM [Ref. 12] and CRYSTAL [Ref. 12] simulations on both the MacPitts and Monterey versions of each circuit. ESIM is an event driven switch level simulator. It was used to verify the logical integrity of the Monterey designs. A successful test was one in which the results obtained from a Monterey design matched the results of the MacPitts version exactly. CRYSTAL performs timing analysis by measuring critical path transmission delays.

ESIM results verified that the new pad router maintained the circuit's logical integrity. Results from Monterey compiler circuits matched those obtained by MacPitts' circuits exactly with one exception. ESIM had difficulty simulating both the MacPitts and Monterey versions of TAXI when the circuit was fitted with tri-state pads. Each version produced different and incorrect results.

Of the many parameters available for performance analysis, wire length, layer type, chip size, number of vias and critical path speed were selected because of their direct relation with the issues which this thesis undertook to investigate; reduction of total chip area and increased chip speeds. Chip size has been widely used as a measure of technology performance. J. Wyatt [Ref 20] demonstrated the contribution that long wire lengths make to signal propagation delays. Table 2, obtained from [Ref. 1] shows that signal delays associated with polysilicon are 100 times the delays experienced with metal1 wires of equal length.

TABLE 1: STATISTICS FOR MACPITTS AND MONTEREY CHIP DESIGNS

CIRCUIT	PARAMETER	MACPITTS	MONTEREY	$\frac{Monterey}{MacPitts} \times 100$
MEMORY				
	length (mm)	1.78	1.67	94
	width (mm)	1.74	1.43	82
	area $(mm)^2$	3.10	2.39	77
	no. vias	60	9	15
	wire length (μ)	17.4×10^3	4.5×10^3	26
	polysilicon (μ)	3.6×10^3	1.1×10^3	31
	metal1 (μ)	13.8×10^3	3.4×10^3	25
	critical path (ns)	258	147	57
TEST				
	length (mm)	2.57	2.04	79
	width (mm)	1.83	1.86	102
	area $(mm)^2$	4.70	3.79	81
	no. vias	65	10	15
	wire length (μ)	28.78×10^3	15.38×10^3	53
	polysilicon (μ)	9.95×10^3	2.04×10^3	20
	metal1 (μ)	18.83×10^3	7.69×10^3	41
	critical path (ns)	671	604	90
TAXI				
	length (mm)	4.07	3.49	86
	width (mm)	2.72	2.81	103
	area $(mm)^2$	11.07	9.81	89
	no. vias	117	15	13
	wire length (μ)	76.4×10^3	38.0×10^3	50
	polysilicon (μ)	25.4×10^3	13.2×10^3	52
	metal1 (μ)	51.0×10^3	24.8×10^3	49
	critical path (ns)	1998	1864	93
MULTIP4				
	length (mm)	5.19	4.70	91
	width (mm)	2.40	2.42	101
	area $(mm)^2$	12.46	11.37	91
	no. vias	104	16	15
	wire length (μ)	57.26×10^3	42.13×10^3	74
	polysilicon (μ)	20.02×10^3	2.73×10^3	14
	metal1 (μ)	37.24×10^3	39.40×10^3	106
	critical path (ns)	3061	2266	74

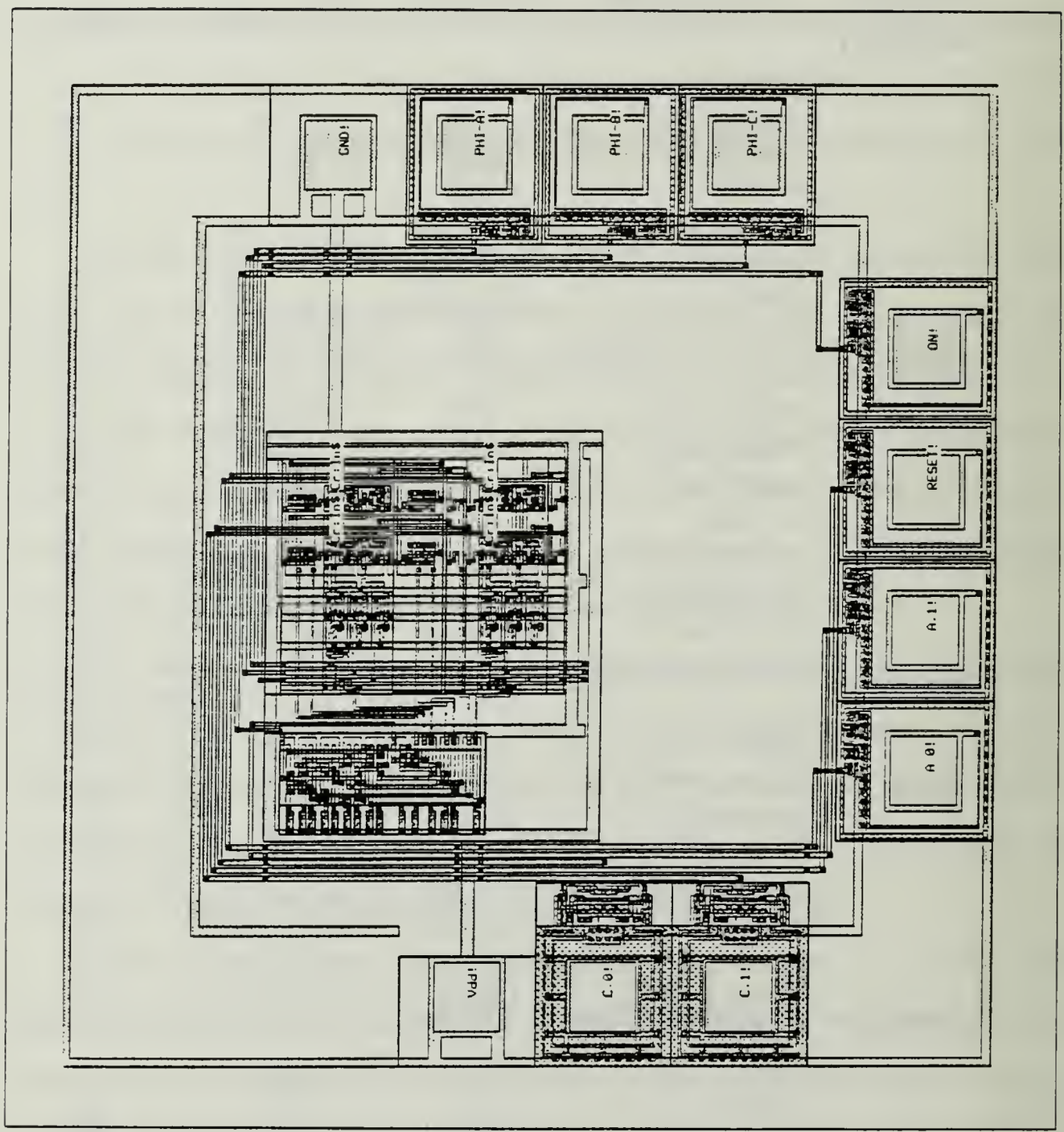

Figure 38: MEMORY circuit design by MacPitts

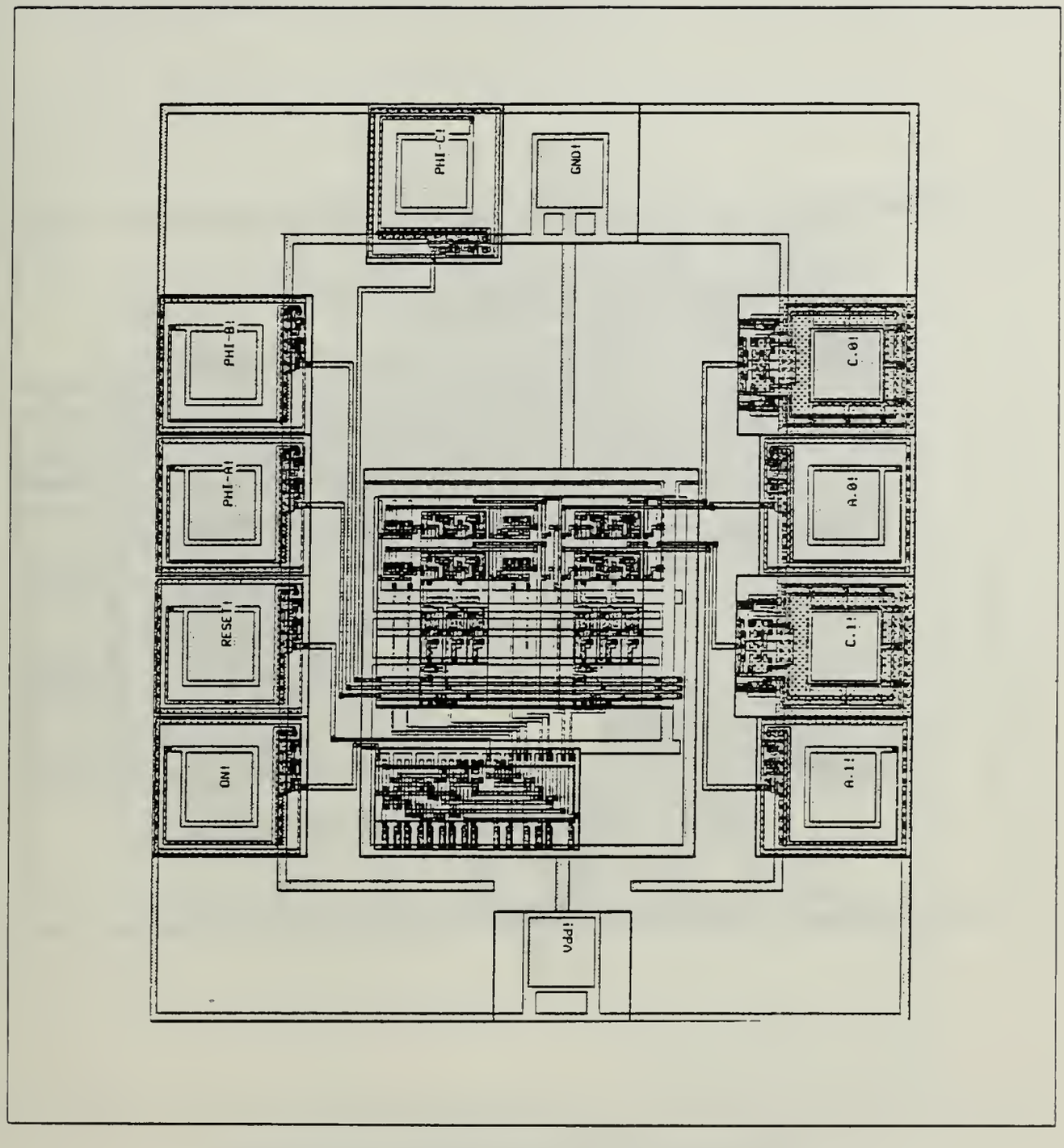

Figure 39: MEMORY circuit design by Monterey

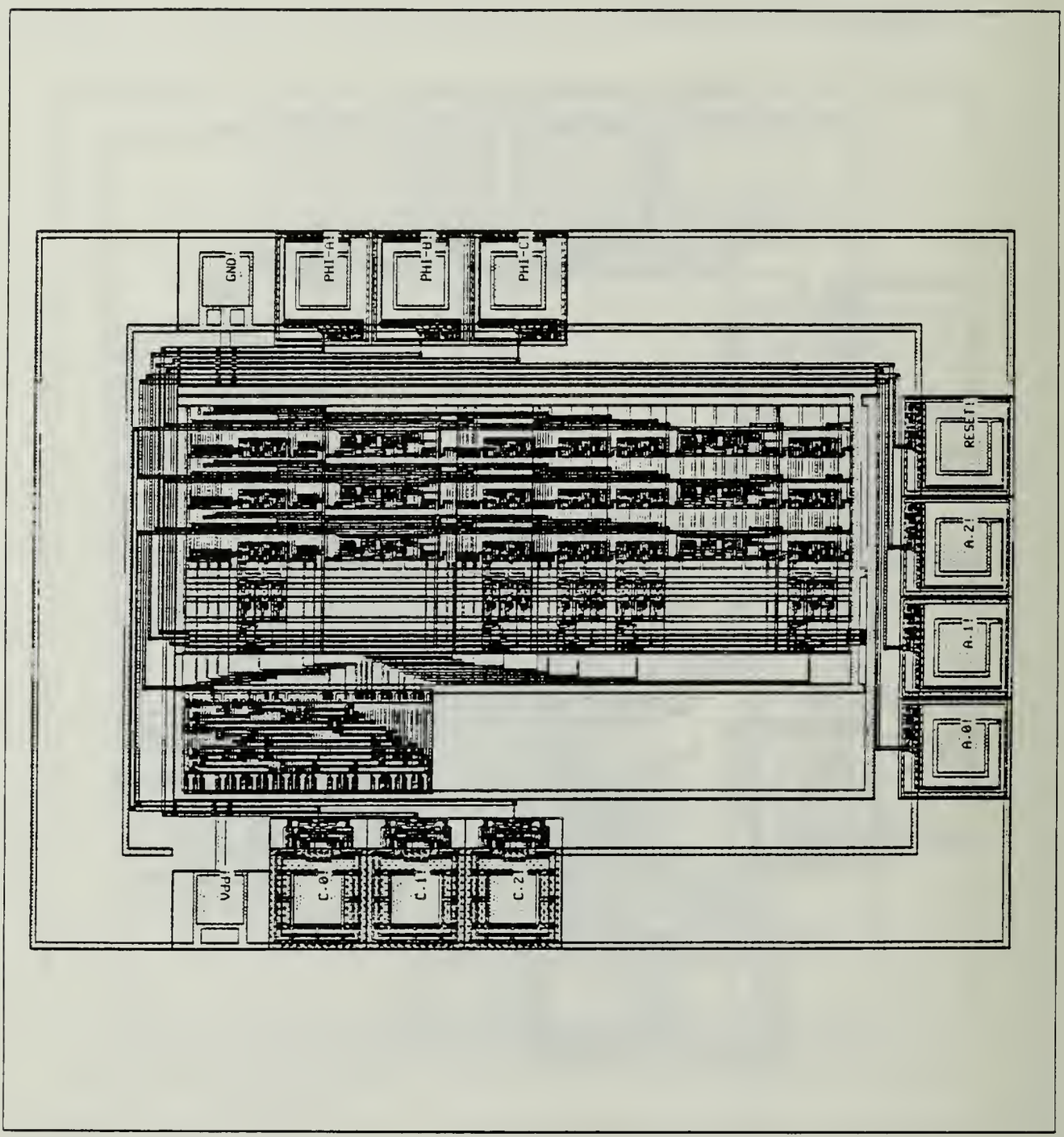

Figure 40: TEST circuit design by MacPitts

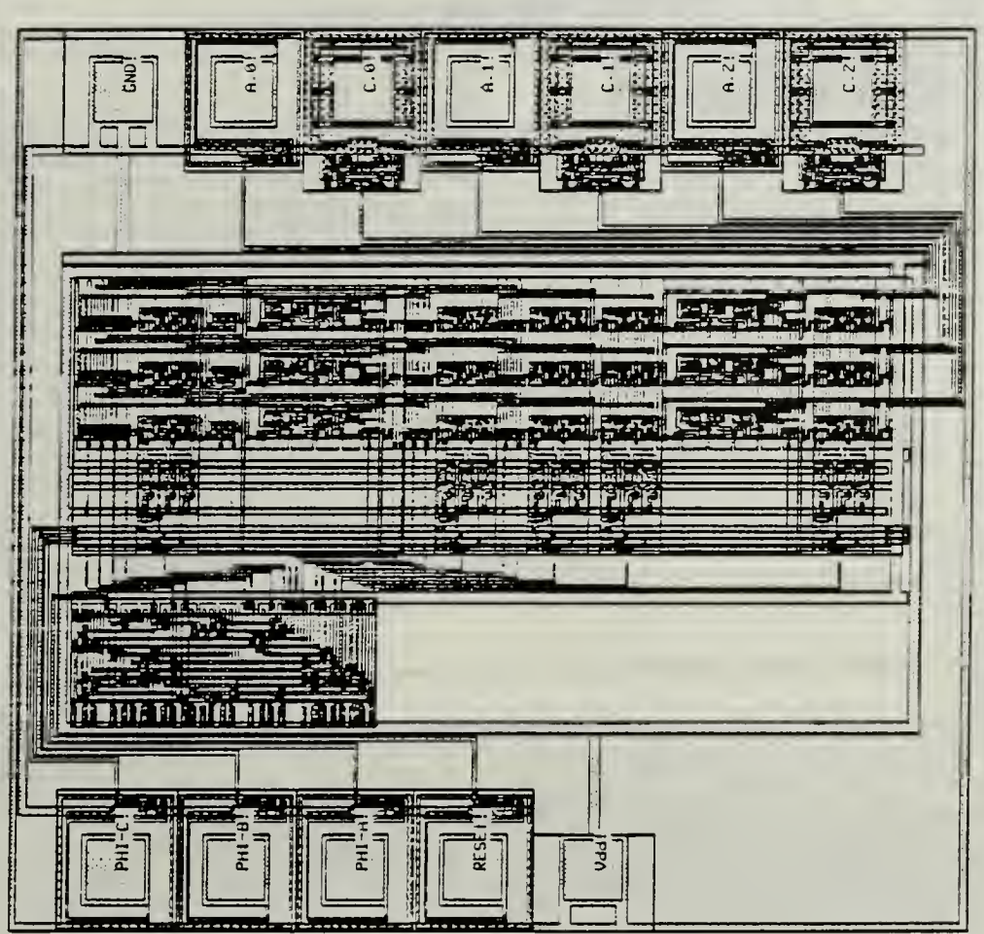

Figure 41: TEST circuit design by Monterey

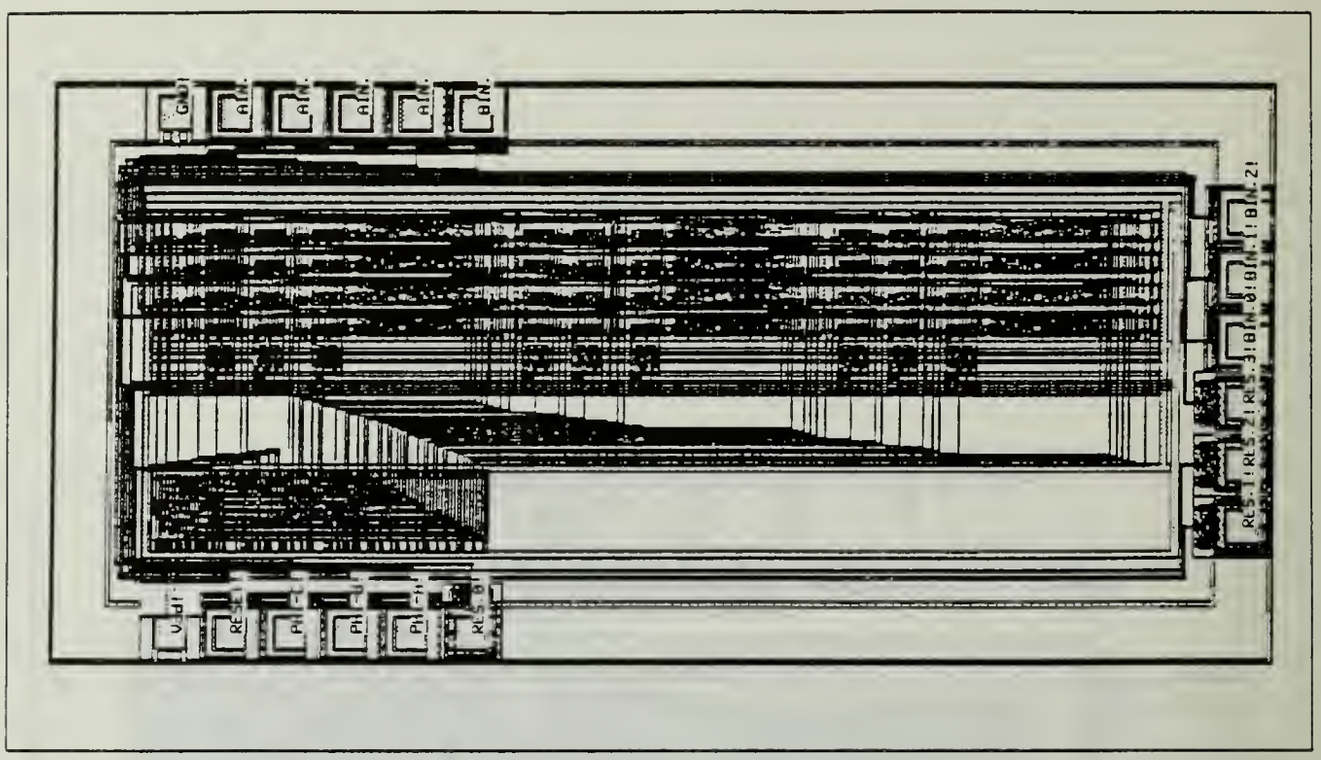

Figure 42: MULTIP4 circuit design by MacPitts

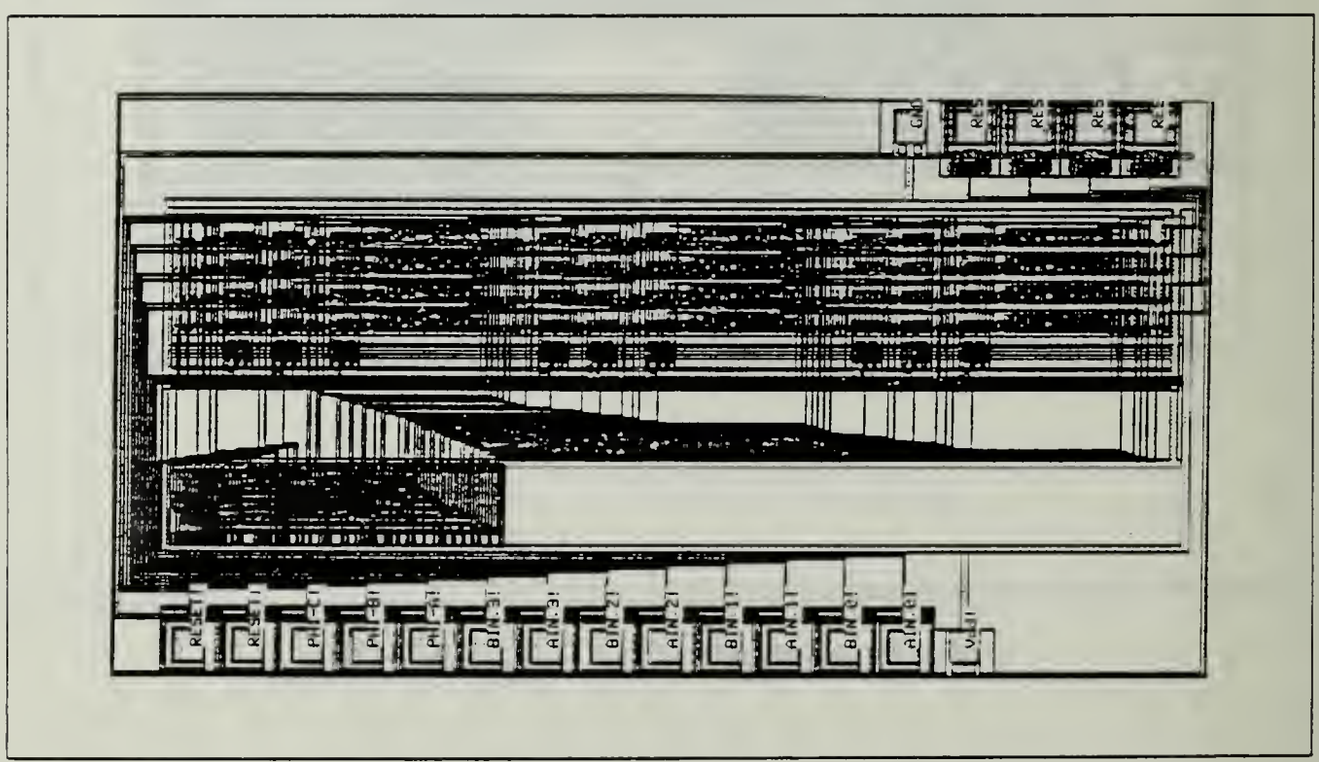

Figure 43: MULTIP4 circuit design by Monterey

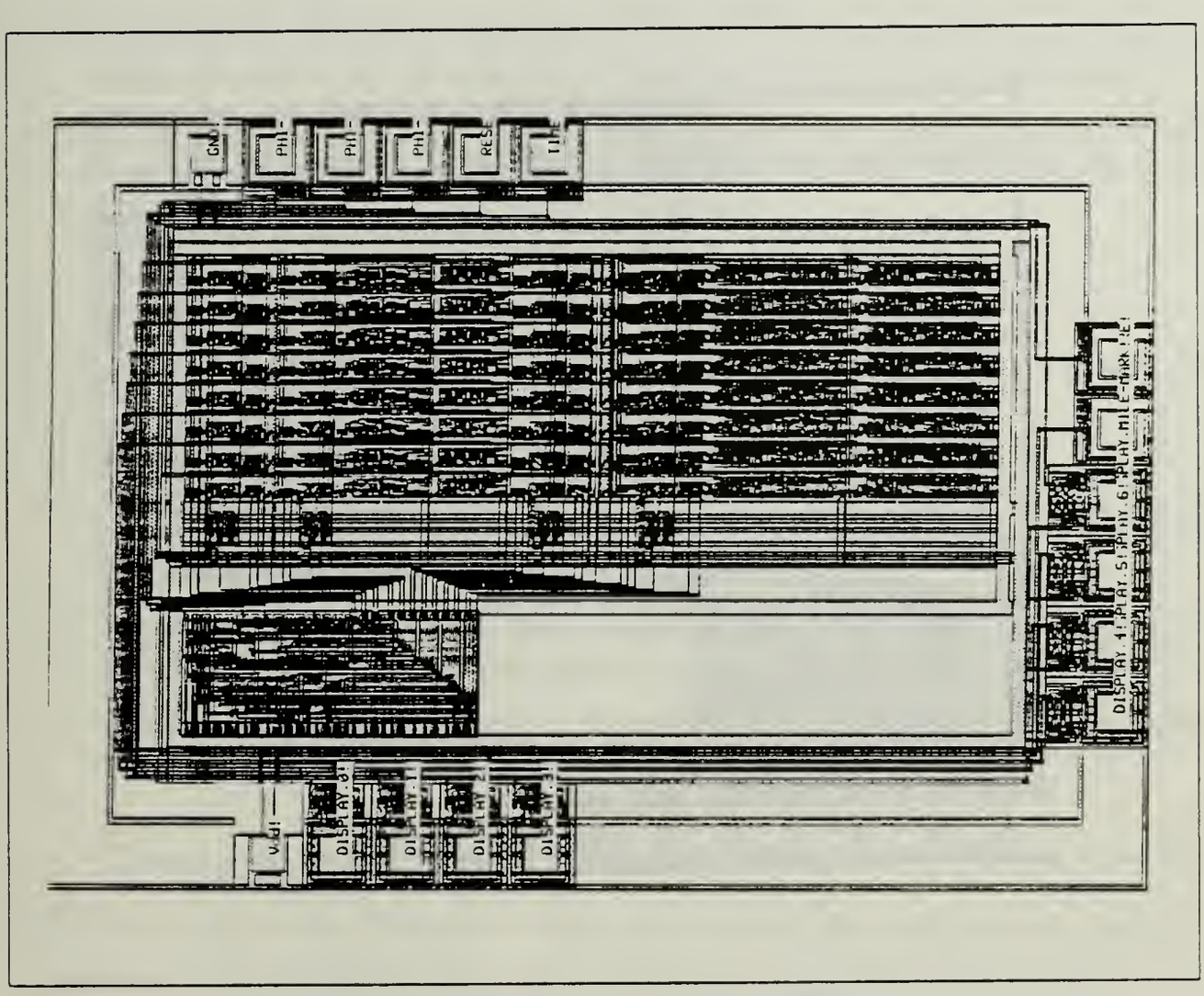

Figure 44: TAXI circuit design by MacPitts

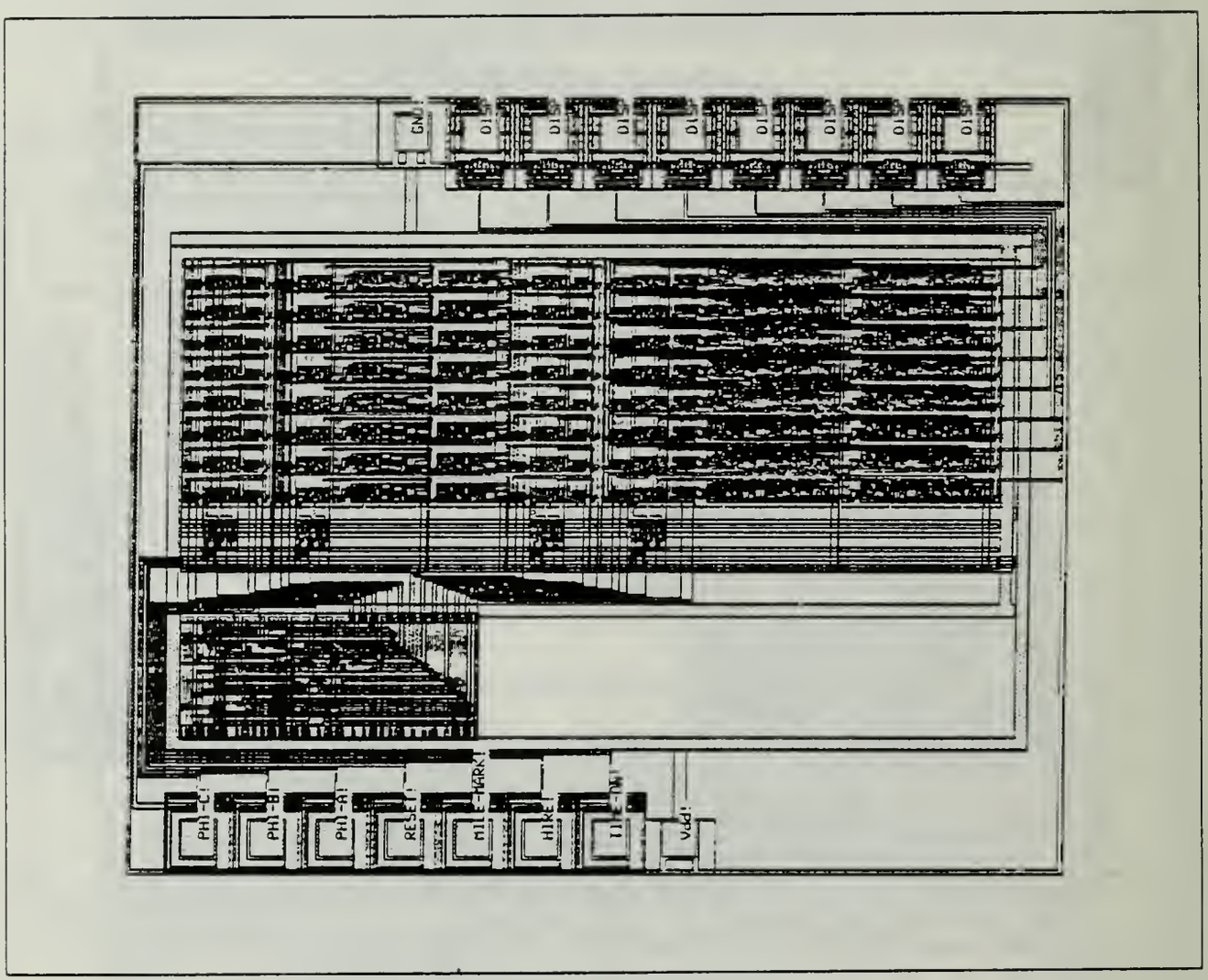

Figure 45: TAXI circuit design by Monterey

1. Area

The MacPitts pad router impacts on the final chip area in three ways:

1. Circuit extension in either the horizontal and, or vertical direction if any of the three allowed pad placement sides cannot fit the number of pads indicated by *number-pins-per-side*. This is common in small circuits with limited placing space. This is the mechanism that produces the large empty areas in MacPitts' version of MEMORY.
2. Since all nets must be routed through the left side, net congestion produces wider routing channels. By necessity, the sum of tracks in the top and bottom channels must equal the number of pads (except for ground and power).
3. Designs reserve space for pads on all four sides yet pads are placed on only three. This is illustrated in Figure 43 where a distance wide enough to accommodate pads is evident on the left side.

The Monterey pad router solves the problems listed above. As a result, it will always produce smaller layouts than MacPitts. Specifically, the Monterey pad router will:

1. Place pads on 2, 3, or 4 sides to minimize area. If the number of pads do not fit around the circuit, the longest side is extended until they do. By extending the longest side, the total increase in area is minimized.
2. Ability to enter data-path from both the left and right sides reduces net congestion on the left side and tends to reduce routing channel widths.
3. The exterior pad power ring is collapsed on sides without pads. This is demonstrated on the right and left sides of Figure 44.

These mechanisms interact to make all Monterey circuits smaller than their MacPitts counterparts. Best performance is obtained for small circuits, under the influence of the first mechanism. For such circuits, area reductions of 20% - 25% are common. Larger circuits exhibit area reductions in the 10% - 15% range. These circuits tend to place all their pads on the bottom and top sides. The third area reduction mechanism is most significant here since both the left and right pad power rings can be collapsed.

2. Wire Length

Wire length is a useful indicator to determine router effects on circuit speed. J. Wyatt [Ref. 20] describes the effect of interconnect on signal delays. The impact

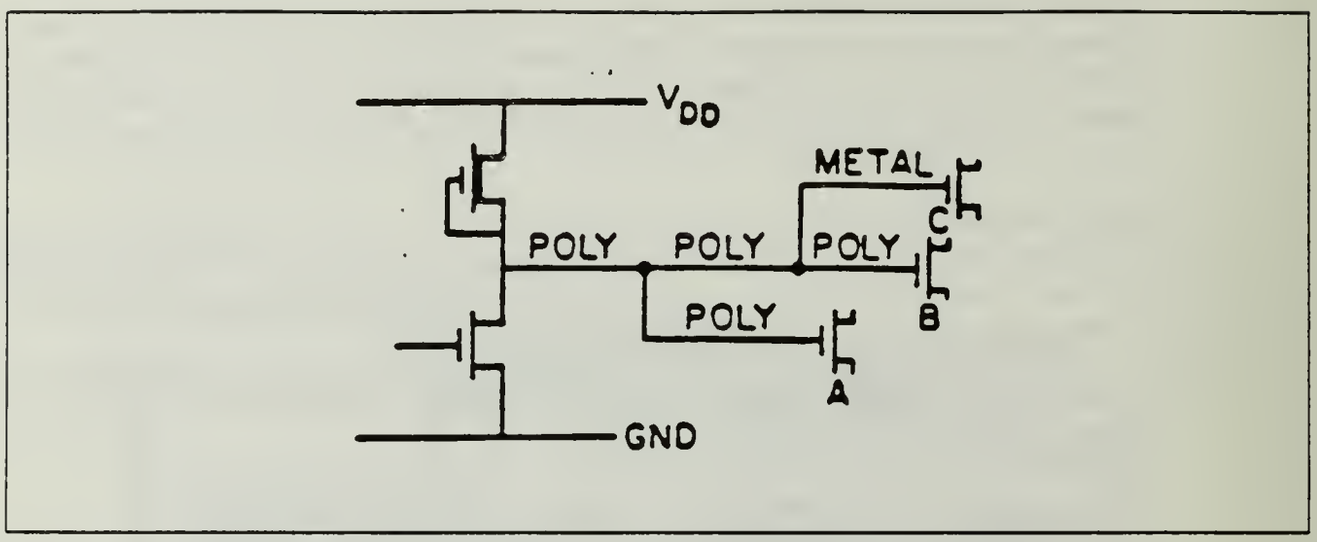

Figure 46: MOS inverter circuit with fanout

TABLE 2: GUIDELINES FOR IGNORING RC WIRE DELAYS [Ref. 1: Table 4.7, pg. 136]

Layer	Maximum Length
Metal	$20,000\lambda$
Silicide	$2,000\lambda$
Polysilicon	200λ
Diffusion	20λ

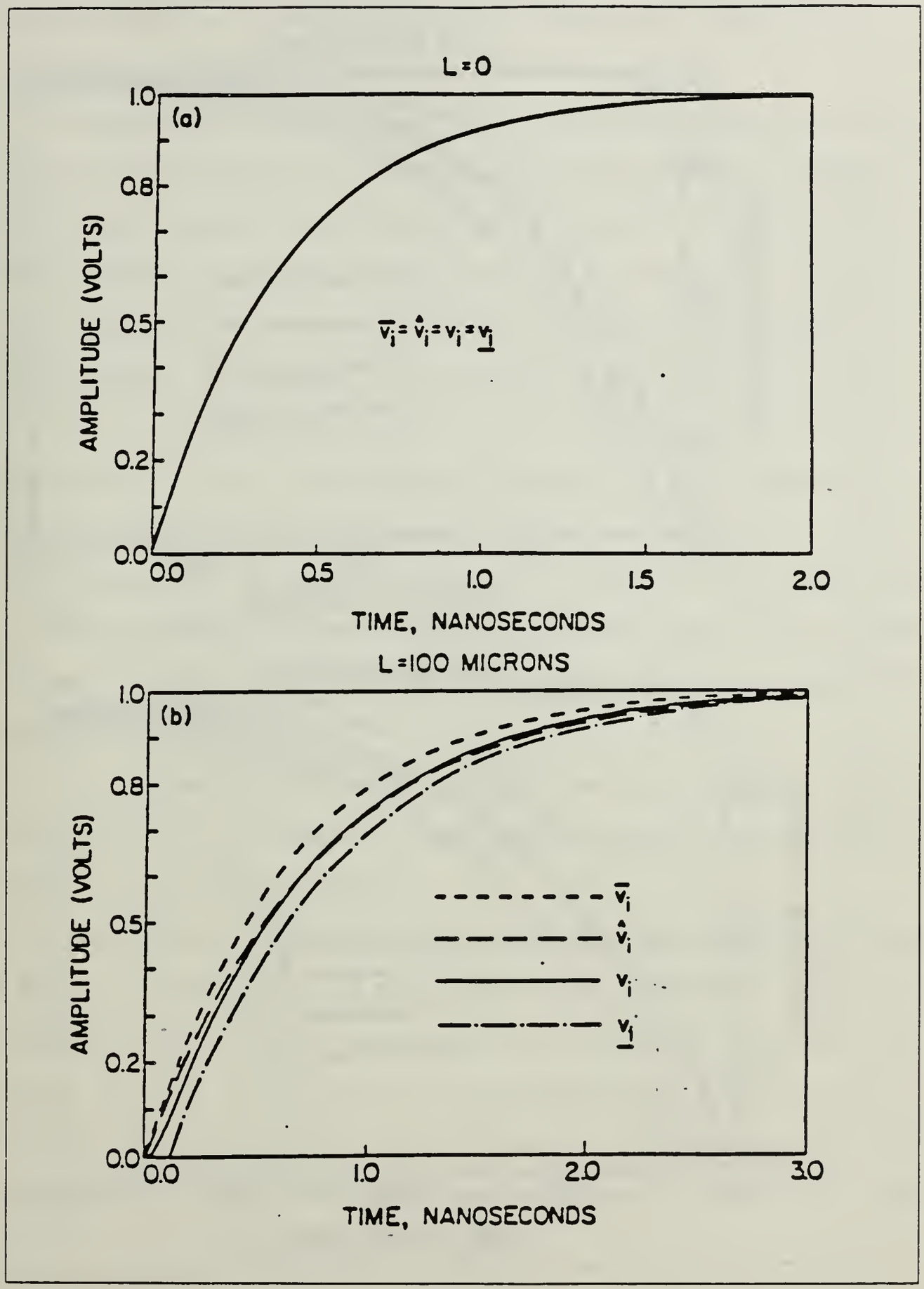

Figure 47: Bounds for the step response of the circuit in Figure 46. (a) $L = 0$, (b) $L = 100\mu$ [Ref. 20: Fig. 11.2.24]

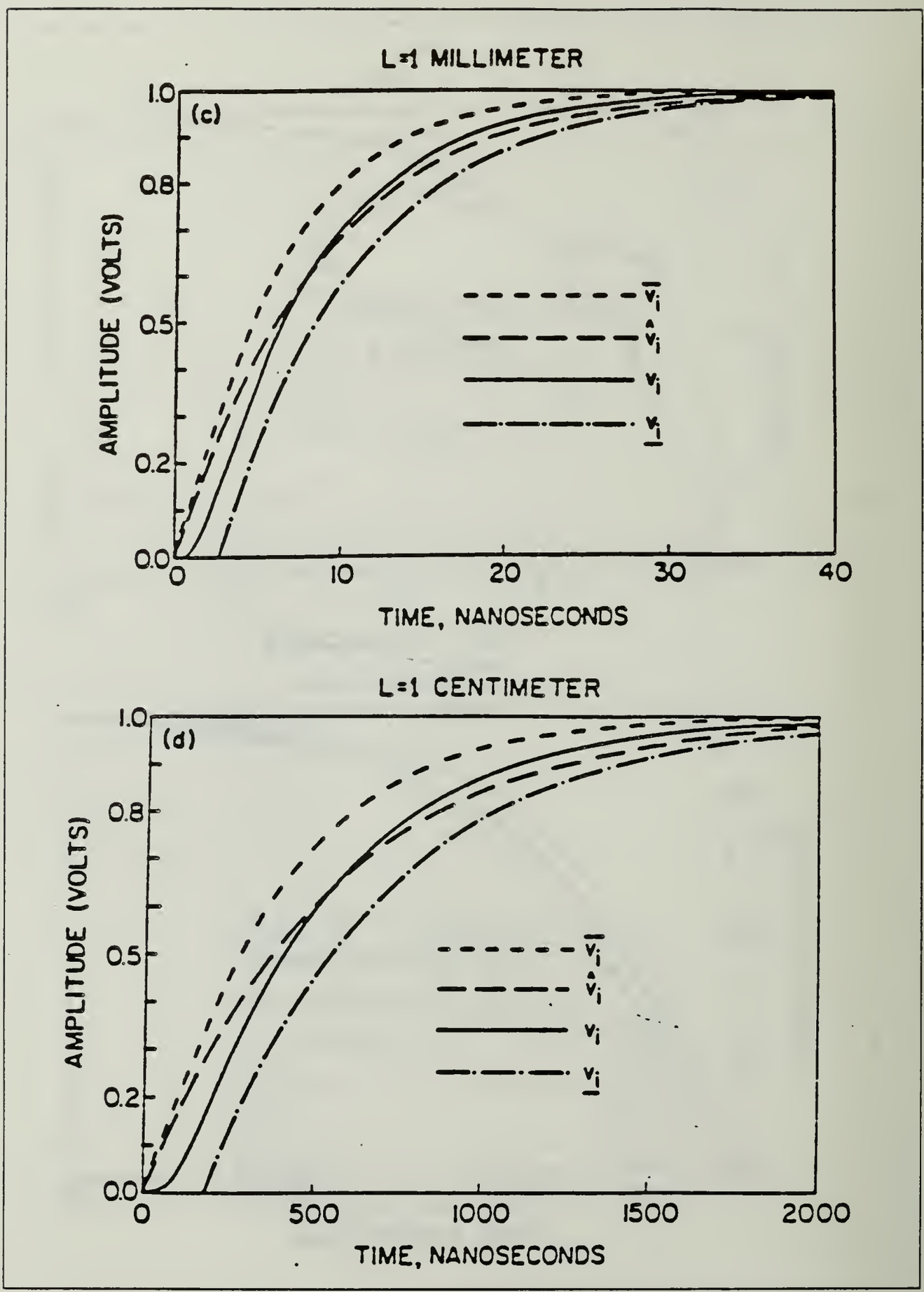

Figure 48: Bounds for the step response of the circuit in Figure 46. (a) $L = 1mm$, (b) $L = 1cm$ [Ref. 20: Fig. 11.2.24]

of varying lengths of polysilicon on the circuit of Figure 46 is shown in Figures 47 and 48. Table 2 shows guidelines for ignoring RC wire delays.

In this analysis, total wire length is the sum of all wires required to connect every net from their pads to their connection points in the data-path, controller or clocks. Thus, wires deep inside data-path may be considered in this measurement. This is necessary because even though these wires are not produced by the pad router, their lengths are definitely influenced by the modified pad placement and routing package. For example, the wire from pad C.2 in Figure 41 must connect to a point on the right edge of data-path. In MacPitts, this net is first routed around the circuit body to the left edge of data-path. From here, a long polysilicon wire is produced by the bus mechanism in data-path to make the final connection to the terminal near the right edge of data-path. A similar situation occurs with net RES.3 in Figure 43. The pad is approximately $0.3mm$ from its terminal inside data-path. MacPitts requires over $9mm$, half of that in polysilicon, to complete the connection.

As the values in Table 2 suggest, the layer used in routing is as important, if not more so, than wire length. To assist with the analysis, total wire lengths in polysilicon and metal1 were obtained.

The Monterey compiler outperformed MacPitts in all circuits, ranging from a 6% improvement in TAXI, to a 43% improvement in MEMORY. Time delay reductions were most pronounced in those circuits with connection points at the far right of data-path. In routing these circuits (MEMORY, MULTIP4 and TEST1) MacPitts produces an extremely long polysilicon wire that spans across the data-path. In contrast, the Monterey silicon compiler accesses these points with a short polysilicon wire on the right edge of data-path.

3. Vias

Vias, or polysilicon to metal cuts are undesirable from the standpoint of circuit performance. The high capacitance associated with the metal to poly interface should be avoided where possible. A discussion of the via reduction issues in the Monterey silicon compiler was presented in Chapter IV.

VI. CONCLUSIONS

A. SUMMARY

This thesis has introduced a new and improved pad router for use by the Monterey silicon compiler to replace the router provided by the MacPitts silicon compiler. The original MacPitts pad router suffered from a very inefficient pad placement and routing algorithms, as well as limitations in the ways signals could be routed to the data-path. The improved performance of the Monterey compiler is the result of:

1. The ability to connect to data-path from either the left or right sides. MacPitts was allows connections to data-path on the left side only.
2. The ability to place pads on 2, 3, or 4 sides. All circuits designed by MacPitts place pads on three sides.
3. Minimize use of polysilicon and vias in the routing area. MacPitts requires up to 6 vias per net compared to one via per net in the Monterey compiler.

The new router was tested with various circuits. Comparative analysis with MacPitts' layouts showed that the Monterey router results in smaller and faster layouts for all circuits. Area improvement ranged from 5% to 20%. Small circuits tend to benefit most in area reduction. Large circuits show modest area improvements, primarily due to the collapse of the outer power ring on sides without pads.

All Monterey routed designs performed faster than their MacPitts' counterparts. The greatest improvement was in circuits requiring connections from pads to the right side of the data-path. Since MacPitts access to the data-path from the left side only, it uses a long polysilicon wire to reach from the terminal inside data-path to the left edge of the circuit. This wire is not required by the Monterey compiler because it can access the terminal through either the left or right sides.

B. RECOMMENDATIONS

The Monterey silicon compiler provdes fertile ground for continued study and development. The following recommendations should be considered for future thesis research:

1. Significant speed improvements are possible in SCMOS technology by routing the long polysilicon wires produced by data-path and by the river router, between the data-path and the controller, with metal1.

2. The fixed floor plan used by MacPitts wastes area. This can be seen by the large empty spaces below the data-path, as well as the area to the right of the controller. A study should be undertaken to examine different cell placement strategies. A possible alternative is to allow the different cells (data-path, controller, flags and pads) to float until an optimal configuration is found. A channel router, in conjunction with a global router could be used to route the cells.

3. An alternative pad router should be developed to create designs that fit the various MOSIS standard chip frames.

APPENDIX A

MACPITTS' FUNCTIONS

This Appendix contains those functions in MacPitts that have a role in pad placement and routing. The first function, **get-basic-buses-from-port-output-unit** is found in the general.l file. All other functions are in frame.l.

```
;;;layout routines

(def get-basic-buses-from-port-output-unit
 (lambda (number-of-units tail unit unit-number)
  (update-basic-buses
   (update-basic-buses
    tail
    (make-port-output-id (port-output-unit-name unit))
    (make-left-tip))))
   (make-port-output-id (port-output-unit-name unit))
   (make-output-tip unit-number))))


(declare (special gates top-part))

(def layout-object
 (lambda (object)
  (prog (definitions flags data-path control pins gates straps
         conductivity power data-path-length control-length
         flags-length top-width bottom-width data-path-layout
         control-layout flags-layout river-layout wing-layout
         skeleton-layout internal-layout pins-layout ring-layout
         layout nets ring-width top-part bottom-part top-bank
         bottom-bank river-width bottom-part-river-points
         intended-right intended-top extended-right extended-top)
(setq definitions (object-definitions object))
   (setq flags (object-flags object))
   (setq data-path (object-data-path object))
   (setq control (object-control object))
   (setq pins (object-pins object))
(herald "Extruding gates")
   (setq gates (extrude-gates control flags))
   (statistic (concat "Control has " (length gates) " columns"))
    (cond ((member? 'opt-c option-list)
           (setq gates
            (nthelem-list
             (order (extrude-basic-straps gates)
                    gates
                    (count (length gates))
                    (function junction-gate-number)
                    (lambda (basic-strap) basic-strap)
                    (lambda (gate1 gate2)
```

```
                              (gate-before? gate1 gate2 gates))
                 (lambda (gate1 gate2)
                              (gate-after? gate1 gate2 gates)))
                         gates))))
   (setq gates (insert-nor-ground-lines gates))
(herald "Extruding straps")
   (setq straps (extrude-straps gates))
    (statistic
    (concat "Circuit has "
            (slash-alpha
              (list (flags-transistor-count flags)
                    (data-path-transistor-count data-path
                                  definitions)
                    (control-transistor-count gates straps)
               (pins-transistor-count pins))
               0
              (function +)
              (lambda (x) (+ (car x) (cadr x))))
             " transistors"))
      (statistic
       (concat "Control has "
               (slash-alpha straps 0 (function max)
                              (function strap-track-number))
          " tracks"))
   (setq conductivity (plus (data-path-conductivity
                               data-path definitions)
                            (control-conductivity gates straps)
                       (flags-conductivity flags)))
   (setq power (conductivity-to-power-bus-width conductivity 11))
(statistic (concat "Power consumption is "
                      (conductivity-to-power-consumption
                       (plus conductivity
                             (pins-conductivity pins)))
                      " Watts"))
   (setq data-path-length
         (max (data-path-required-length data-path definitions)
                                    4))
   (setq control-length (control-required-length gates))
   (setq flags-length (max (flags-required-length flags power)
                     4))
   (setq top-width (max (data-path-required-width
                           data-path power definitions)
                         (flags-required-width flags power)))
(setq bottom-width (control-required-width straps))
(herald "Laying out data-path")
   (setq data-path-layout
         (layout-data-path data-path power top-width
         definitions))
(herald "Laying out control")
   (setq control-layout
         (layout-control gates straps power bottom-width))
(herald "Laying out flags")
   (setq flags-layout (layout-flags flags power top-width))
(herald "Laying out river")
```

```
   (setq top-part
    (merge (move data-path-layout (+ power 3) 0)
           (move flags-layout
             (+ power 3 data-path-length 3 power 3) 0)))
   (setq bottom-part
         (move control-layout (+ power 3) (- power 4)))
   (setq bottom-part-river-points
        (find-attributes bottom-part '(river)))
   (setq top-bank
         (sort (alpha (lambda (point)
                        (point-x (find top-part
                                       (point-name point))))
                      bottom-part-river-points)
                      (function <)))
    (setq bottom-bank
          (sort (alpha (function point-x)
                       bottom-part-river-points)
                (function <)))
   (setq river-width
         (+ (river-span 'NP 2 top-bank bottom-bank)
                (wing-span bottom-part)
                (- 4 power)))
   (setq intended-top
     (+ power bottom-width power river-width (driver-width)
   power top-width power 3 power))
   (setq intended-right
    (+ power 3
       (max control-length (+ data-path-length 3 power
                              3 flags-length))
       3 power))
   (setq river-layout
    (river 'NP 2 (wing-span bottom-part) top-bank bottom-bank))
(herald "Laying out wing")
   (setq wing-layout
         (layout-wing (sort (find-attributes
                               bottom-part '(wing))
                            (flambda (point1 point2)
                               (< (point-x point1)
                                      (point-x point2))))))
 (herald "Laying out skeleton")
   (setq skeleton-layout
    (layout-skeleton power intended-top intended-right
                     data-path-length bottom-width river-width))
   (setq internal-layout
    (merge
     (move top-part
           0 (+ power bottom-width power river-width
                (driver-width) power))
           bottom-part
           (move (rotcw river-layout)
                 0 (+ power bottom-width power river-width))
 (move wing-layout 0 (+ power bottom-width 4))
           skeleton-layout))
(herald "Laying out pins")
```

```
   (setq pins-layout
         (layout-pins
          pins
          power
          intended-right
          intended-top
          (make-ring-width 0 0 0 0)
          (lookup-logo definitions)))
   (setq extended-right (extend-right pins intended-right))
   (setq extended-top (extend-top pins intended-top))
   (setq ring-width
         (get-ring-width
          (merge internal-layout pins-layout) extended-right
                                              extended-top))
   (setq pins-layout
         (layout-pins
          pins
          power
          intended-right
          intended-top
          ring-width
          (lookup-logo definitions)))
   (setq layout
         (first-quadrant (merge internal-layout pins-layout
                                ring-layout)))
    (statistic (concat "Dimensions are "
       ;;jh replaced minimum-feature-size with lambda-spacing.
                  (quotient (times (right layout)
                                           (lambda-spacing))
                                   100000.0)
                        " mm by "
                        (quotient (times (top layout)
                                           (lambda-spacing))
                                   100000.0)
                        " mm"))
   (return layout))))}

(def wing-span
 (lambda (item)
  (1+ (* 5 (length (find-attributes item '(wing)))))))

(defsymbol layout-wing (points)
 (merge-list
  (alpha
   (lambda (point wing-number)
    (merge
     (rect 'NP 0
               (- (* 5 wing-number) 3)
               (+ (point-x point) 1)
               (- (* 5 wing-number) 1))
     (rect 'NP (- (point-x point) 1)
               0
               (+ (point-x point) 1)
               (- (* 5 wing-number) 1))
```

```
     (mark (car (point-name point))
           0
           (- (* 5 wing-number) 2)
           'NP '
           (ring left inside))))
     points
     (count (length points)))))


;;;layout nets

(declare (special nets right top power power-point ground-point)) (def
layout-nets
 (lambda (nets right top power power-point ground-point)
  (merge-list
   (alpha (lambda (net)
           (layout-net net nets right top power power-point
                       ground-point))
          nets))))

(declare (unspecial nets right top power
                    power-point ground-point))
(def layout-net
 (lambda (net nets right top power power-point ground-point)
  (cond ((is-point-top? (car (net-basic-net net)))
         (layout-top-net net nets right top
                         (cond ((is-point-top? power-point)
                           (point-x power-point))
                               ((is-point-top? ground-point)
                                (point-x ground-point))
                               (t ()))
                         power))
        ((is-point-right? (car (net-basic-net net)))
         (layout-right-net net nets right top
                         (cond ((is-point-right? power-point)
                                (point-y power-point))
                               ((is-point-right? ground-point)
                                (point-y ground-point))
                                (t ()))
                         power))
        ((is-point-bottom? (car (net-basic-net net)))
           (layout-bottom-net net nets right top
                           (cond ((is-point-bottom? power-point)
                             (point-x power-point))
                                 ((is-point-bottom?
                                     ground-point)
                                     (point-x ground-point))
                                 (t ()))
                            power))
        ((is-point-left? (car (net-basic-net net)))
         (layout-left-net net nets right top
                         (cond ((is-point-left? power-point)
                                (point-y power-point))
                               ((is-point-left? ground-point)
```

```
                                (point-y ground-point))
                               (t ()))
                         power)))))

(declare (special nets right top track-number))

(def layout-top-net
 (lambda (net nets right top skip power)
  (let ((basic-net (net-basic-net net))
        (track-number (net-track-number net)))
    (let ((left-x
           (cond ((is-point-first? (basic-net-left-point
                                    basic-net)) 2)
           (t (point-x (basic-net-left-point basic-net)))))
    (right-x
           (cond ((is-point-last? (basic-net-right-point
                                   basic-net))
                  (- right 2))
                 (t (point-x (basic-net-right-point
                              basic-net))))))
     (merge
      (merge-list
      (alpha (lambda (point)
                  (layout-top-point point nets right
                                    top track-number))
             basic-net))
       (cond
           ((or (null skip)
           (< right-x (- skip (/up power 2) 3))
           (> left-x (+ skip (/up power 2) 3)))
       (rect 'NM left-x
                 (+ top (* 7 track-number) -4)
                 right-x
                 (+ top (* 7 track-number))))
        (t (merge
            (rect 'NM left-x
                      (+ top (* 7 track-number) -4)
                      (- skip (/up power 2) 3)
                      (+ top (* 7 track-number)))
            (rect 'NM (+ skip (/up power 2) 3)
                      (+ top (* 7 track-number) -4)
                      right-x
                      (+ top (* 7 track-number)))
            (rect 'NP (- skip (/up power 2) 7)
                      (+ top (* 7 track-number) -4)
                      (+ skip (/up power 2) 7)
                      (+ top (* 7 track-number)))
            (move (poly-cut) (- skip (/up power 2) 7)
                             (+ top (* 7 track-number)))
            (move (poly-cut) (+ skip (/up power 2) 3)
                             (+ top (* 7 track-number)))))))))))

(declare (unspecial nets right top track-number))
```

```
(declare (special nets right top track-number))

(def layout-right-net
 (lambda (net nets right top skip power)
  (let ((basic-net (net-basic-net net))
        (track-number (net-track-number net)))
   (let ((top-y
          (cond ((is-point-first? (basic-net-left-point
                                   basic-net)) top)
                (t (point-y (basic-net-left-point basic-net)))))
    (bottom-y
          (cond ((is-point-last? (basic-net-right-point
                                  basic-net)) 0)
                (t (point-y (basic-net-right-point
                             basic-net))))))
     (merge
      (merge-list
       (alpha (lambda (point)
                      (layout-right-point point nets right
                                          top track- number))
              basic-net))
       (cond
        ((or (null skip)
             (< top-y (- skip (/up power 2) 3))
             (> bottom-y (+ skip (/up power 2) 3)))
         (rect 'NM (+ right (* 7 track-number) -4)
                   bottom-y
                   (+ right (* 7 track-number))
                   top-y))
        (t (merge
            (rect 'NM (+ right (* 7 track-number) -4)
                      bottom-y
                      (+ right (* 7 track-number))
                      (- skip (/up power 2) 3))
            (rect 'NM (+ right (* 7 track-number) -4)
                      (+ skip (/up power 2) 3)
                      (+ right (* 7 track-number))
                      top-y)
            (rect 'NP (+ right (* 7 track-number) -4)
                      (- skip (/up power 2) 7)
                      (+ right (* 7 track-number))
                      (+ skip (/up power 2) 7))
            (move (poly-cut) (+ right (* 7 track-number))
                             (- skip (/up power 2) 7))
            (move (poly-cut) (+ right (* 7 track-number))
                             (+ skip (/up power 2) 3)))))))))))

(declare (unspecial nets right top track-number))

(declare (special nets right top track-number))

(def layout-bottom-net
 (lambda (net nets right top skip power)
  (let ((basic-net (net-basic-net net))
```

```
        (track-number (net-track-number net)))
   (let ((right-x
          (cond ((is-point-first? (basic-net-left-point
                                   basic-net))
                 (- right 2))
                (t (point-x (basic-net-left-point basic-net)))))
         (left-x
          (cond ((is-point-last? (basic-net-right-point
                                  basic-net)) 2)
                (t (point-x (basic-net-right-point
                             basic-net))))))
     (merge
      (merge-list
       (alpha (lambda (point)
                      (layout-bottom-point point nets right
                                           top track-number))
              basic-net))
       (cond
        ((or (null skip)
             (< right-x (- skip (/up power 2) 3))
             (> left-x (+ skip (/up power 2) 3)))
         (rect 'NM left-x
                   (- (* 7 track-number))
                   right-x
                   (- 4 (* 7 track-number))))
        (t (merge
             (rect 'NM left-x
                       (- (* 7 track-number))
                       (- skip (/up power 2) 3)
                       (- 4 (* 7 track-number)))
             (rect 'NM (+ skip (/up power 2) 3)
                       (- (* 7 track-number))
                       right-x
                       (- 4 (* 7 track-number)))
             (rect 'NP (- skip (/up power 2) 7)
                       (- (* 7 track-number))
                       (+ skip (/up power 2) 7)
                       (- 4 (* 7 track-number)))
             (move (poly-cut) (- skip (/up power 2) 7)
                              (- 4 (* 7 track-number)))
             (move (poly-cut) (+ skip (/up power 2) 3)
                              (- 4 (* 7 track-number)))))))))))

(declare (unspecial nets right top track-number))

(declare (special nets right top track-number))

(def layout-left-net
 (lambda (net nets right top skip power)
  (let ((basic-net (net-basic-net net))
        (track-number (net-track-number net)))
   (let ((bottom-y
          (cond ((is-point-first? (basic-net-left-point
                                   basic-net)) 0)
```

```
                (t (point-y (basic-net-left-point basic-net)))))
          (top-y
           (cond ((is-point-last? (basic-net-right-point
                                   basic-net)) top)
                 (t (point-y (basic-net-right-point
                              basic-net))))))
      (merge
       (merge-list
        (alpha (lambda (point)
                       (layout-left-point point nets right
                                          top track-number))
               basic-net))
        (cond
         ((or (null skip)
              (< top-y (- skip (/up power 2) 3))
              (> bottom-y (+ skip (/up power 2) 3)))
          (rect 'NM (- (* 7 track-number))
                    bottom-y
                    (- 4 (* 7 track-number))
                    top-y))
         (t (merge
             (rect 'NM (- (* 7 track-number))
                       bottom-y
                       (- 4 (* 7 track-number))
                       (- skip (/up power 2) 3))
             (rect 'NM (- (* 7 track-number))
                       (+ skip (/up power 2) 3)
                       (- 4 (* 7 track-number))
                       top-y)
             (rect 'NP (- (* 7 track-number))
                       (- skip (/up power 2) 7)
                       (- 4 (* 7 track-number))
                       (+ skip (/up power 2) 7))
             (move (poly-cut) (- 4 (* 7 track-number))
                              (- skip (/up power 2) 7))
             (move (poly-cut) (- 4 (* 7 track-number))
                              (+ skip (/up power 2) 3)))))))))

(declare (unspecial nets right top track-number))

(def layout-top-point
 (lambda (point nets right top track-number)
  (cond
   ((is-point-first? point)
    (merge
     (move (poly-cut) (- (* 7
                            (last-point-track-number
                               (point-name point) 'left nets)))
                      (+ top (* 7 track-number)))
     (move (poly-cut) 0
                      (+ top (* 7 track-number)))
     (rect 'NP (- (* 7
                     (last-point-track-number (point-name point)
                                              'left nets)))
```

```
                (+ top -4 (* 7 track-number))
               0
               (+ top (* 7 track-number)))))
  ((is-point-last? point)
   (merge
    (move (poly-cut) (+ right -4
                        (* 7 (first-point-track-number
                                (point-name point)
                                'right nets)))
                     (+ top (* 7 track-number)))
    (move (poly-cut) (- right 4)
                     (+ top (* 7 track-number)))
    (rect 'NP right
              (+ top -4 (* 7 track-number))
              (+ right -4
                 (* 7 (first-point-track-number
                         (point-name point) 'right nets)))
              (+ top (* 7 track-number)))))
  (t (merge
      (move (poly-cut) (- (point-x point) 2)
                       (+ top (* 7 track-number)))
      (rect 'NP (1- (point-x point))
                (min (+ top (* 7 track-number)) (point-y point))
           (1+ (point-x point))
                (max (+ top (* 7 track-number))
                     (point-y point)))))))

(def layout-right-point
 (lambda (point nets right top track-number)
  (cond
   ((is-point-first? point)
    (rect 'NM (+ right (* 7 track-number) -4)
              top
              (+ right (* 7 track-number))
              (+ top (* 7 (last-point-track-number
                            (point-name point) 'top nets)))))
   ((is-point-last? point)
    (rect 'NM (+ right (* 7 track-number) -4)
              (- (* 7 (first-point-track-number
                        (point-name point) 'bottom nets)))
              (+ right (* 7 track-number))
              0))
   (t (merge
       (move (poly-cut) (+ right -4 (* 7 track-number))
                        (+ (point-y point) 2))
       (rect 'NP (min (+ right -4 (* 7 track-number))
                      (point-x point))
                 (1- (point-y point))
                 (max (+ right -4 (* 7 track-number))
                      (point-x point))
                 (1+ (point-y point))))))))

(def layout-bottom-point
 (lambda (point nets right top track-number)
```

```
  (cond
   ((is-point-first? point)
    (merge
     (move (poly-cut) (+ right -4
                         (* 7 (last-point-track-number
                                (point-name point) 'right nets)))
                  (- 4 (* 7 track-number)))
     (move (poly-cut) (- right 4)
                      (- 4 (* 7 track-number)))
     (rect 'NP (- right 4)
               (- (* 7 track-number))
               (+ right (* 7 (last-point-track-number
                               (point-name point) 'right nets)))
               (- 4 (* 7 track-number)))))
   ((is-point-last? point)
    (merge
     (move (poly-cut) (- (* 7 (first-point-track-number
                                (point-name point) 'left nets)))
                      (- 4 (* 7 track-number)))
     (move (poly-cut) 0
                      (- 4 (* 7 track-number)))
     (rect 'NP (- (* 7 (first-point-track-number
                         (point-name point) 'left nets)))
               (- (* 7 track-number))
               0
               (- 4 (* 7 track-number)))))
   (t (merge
       (move (poly-cut) (- (point-x point) 2)
                        (- 4 (* 7 track-number)))
       (rect 'NP (1- (point-x point))
                 (min (point-y point) (- 4 (* 7 track-number)))
                 (1+ (point-x point))
                 (max (point-y point)
                      (- 4 (* 7 track-number)))))))))

(def layout-left-point
 (lambda (point nets right top track-number)
  (cond
   ((is-point-first? point)
    (rect 'NM (- (* 7 track-number))
              (- (* 7 (last-point-track-number (point-name point)
                                               'bottom nets)))
              (- 4 (* 7 track-number))
              0))
   ((is-point-last? point)
    (rect 'NM (- (* 7 track-number))
              top
              (- 4 (* 7 track-number))
              (+ top (* 7 (first-point-track-number
                            (point-name point) 'top nets)))))
   (t (merge
       (move (poly-cut) (- (* 7 track-number))
                        (+ (point-y point) 2))
       (rect 'NP (min (- (* 7 track-number)) (point-x point))
```

```
                (1- (point-y point))
                (max (- (* 7 track-number)) (point-x point))
                (1+ (point-y point)))))))))


(declare (special net-name side))

(def first-point-track-number
 (lambda (net-name side nets)
  (net-track-number
   (first-that
    nets
    ()
    (lambda (net)
     (and (member? side (point-attributes
                         (basic-net-left-point (net-basic-net net))))
          (is-point-first? (basic-net-left-point (net-basic-net net)))
          (equal (point-name (basic-net-left-point
                              (net-basic-net net)))
                 net-name)))))))

(declare (unspecial net-name side))

(declare (special net-name side))

(def last-point-track-number
 (lambda (net-name side nets)
  (net-track-number
   (first-that
    nets
    ()
    (lambda (net)
     (and (member? side
                   (point-attributes
                    (basic-net-right-point
                     (net-basic-net net))))
          (is-point-last? (basic-net-right-point
                           (net-basic-net net)))
          (equal (point-name (basic-net-right-point
                              (net-basic-net net)))
                 net-name)))))))

(declare (unspecial net-name side))

(def get-ring-width
 (lambda (item right top)
  (make-ring-width
   (* 7 (net-track-number
         (minmax (extract-nets item 'top right top)
                 (lambda (net1 net2)
                         (> (net-track-number net1)
                            (net-track-number net2))))))
   (* 7 (net-track-number
         (minmax (extract-nets item 'right right top)
```

```
                (lambda (net1 net2)
                        (> (net-track-number net1)
                           (net-track-number net2))))))
    (* 7 (net-track-number
          (minmax (extract-nets item 'bottom right top)
                  (lambda (net1 net2)
                          (> (net-track-number net1)
                             (net-track-number net2))))))
    (* 7 (net-track-number
          (minmax (extract-nets item 'left right top)
                  (lambda (net1 net2)
                          (> (net-track-number net1)
                             (net-track-number net2)))))))))

(declare (special side))

(def extract-nets
 (lambda (item side right top)
  (allocate-tracks
   (such-that (extract-subnets
               (rotate-basic-nets
                (order-basic-nets (extract-basic-nets item))
                right top))
              (lambda (basic-net)
               (member? side
                        (point-attributes
                        (car basic-net)))))
   (function basic-net-left-point)
   (function basic-net-right-point)
   (function basic-net-point-further-left?)
   (function basic-net-overlap?))))

(declare (unspecial side))

(def extract-subnets
 (lambda (nets)
  (cond
   ((null nets) ())
   (t (append (extract-subnet (car nets))
              (extract-subnets (cdr nets)))))))

(def extract-subnet
 (lambda (net)
  (let ((net-name (point-name (car net))))
   (cond ((null net) ())
         ((is-point-top? (car net))
          (extract-top-subnet (list (car net)) ()
                              (cdr net) net-name))
         ((is-point-right? (car net))
          (extract-right-subnet (list (car net)) ()
                                (cdr net) net-name))
         ((is-point-bottom? (car net))
          (extract-bottom-subnet (list (car net)) ()
                                 (cdr net) net-name))
```

```
        ((is-point-left? (car net))
         (extract-left-subnet (list (car net)) ()
                              (cdr net) net-name))))))

(def extract-top-subnet
 (lambda (subnet subnets net net-name)
  (cond ((null net) (cons subnet subnets))
        ((is-point-top? (car net))
         (extract-top-subnet
          (append1 subnet (car net)) subnets (cdr net) net-name))
   ((is-point-right? (car net))
         (extract-right-subnet
          (list (make-point net-name () () () '
                            (right first ring)))
          (append
           (list (append1 subnet
                          (make-point net-name () () ()
                                      '(top last ring))))
           subnets)
          net
          net-name))
        ((is-point-bottom? (car net))
         (extract-bottom-subnet
          (list (make-point net-name () () ()
                            '(bottom first ring)))
          (append
           (list (append1 subnet
                          (make-point net-name () () ()
                                      '(top last ring)))
                 (list (make-point net-name () () ()
                                   '(right first ring))
                       (make-point net-name () () ()
                                   '(right last ring))))
           subnets)
          net
          net-name))
        ((is-point-left? (car net))
         (extract-left-subnet
          (list (make-point net-name () () ()
                            '(left first ring)))
          (append
           (list (append1 subnet
                          (make-point net-name () () ()
                                      '(top last ring)))
                 (list (make-point net-name () () ()
                                   '(right first ring))
                       (make-point net-name () () ()
                                   '(right last ring)))
                 (list (make-point net-name () () ()
                                   '(bottom first ring))
                       (make-point net-name () () ()
                                   '(bottom last ring))))
            subnets)
          net
```

```
          net-name)))))

(def extract-right-subnet
 (lambda (subnet subnets net net-name)
  (cond ((null net) (cons subnet subnets))
        ((is-point-top? (car net))
         (extract-top-subnet
          (list (make-point net-name () () () '(top first ring)))
     (append
           (list (append1 subnet
                          (make-point net-name () () ()
                                      '(right last ring)))
                 (list (make-point net-name () () ()
                                   '(bottom first ring))
                       (make-point net-name () () ()
                                   '(bottom last ring)))
                 (list (make-point net-name () () ()
                                   '(left first ring))
                       (make-point net-name () () ()
                                   '(left last ring))))
            subnets)
           net
           net-name))
        ((is-point-right? (car net))
         (extract-right-subnet
          (append1 subnet (car net)) subnets (cdr net) net-name))
   ((is-point-bottom? (car net))
         (extract-bottom-subnet
          (list (make-point net-name () () ()
                            '(bottom first ring)))
           (append
            (list
             (append1 subnet (make-point net-name () () ()
                                         '(right last ring))))
            subnets)
           net
           net-name))
        ((is-point-left? (car net))
         (extract-left-subnet
          (list (make-point net-name () () ()
                            '(left first ring)))
          (append
           (list (append1 subnet
                          (make-point net-name () () ()
                                      '(right last ring)))
                 (list (make-point net-name () () ()
                                   '(bottom first ring))
                       (make-point net-name () () ()
                                   '(bottom last ring))))
            subnets)
           net
           net-name)))))

(def extract-bottom-subnet
```

```
(lambda (subnet subnets net net-name)
 (cond ((null net) (cons subnet subnets))
       ((is-point-top? (car net))
        (extract-top-subnet
         (list (make-point net-name () () () '(top first ring)))
    (append
          (list
           (append1 subnet (make-point net-name () () ()
                                       '(bottom last ring)))
           (list (make-point net-name () () ()
                             '(left first ring))
                 (make-point net-name () () ()
                             '(left last ring))))
          subnets)
         net
         net-name))
       ((is-point-right? (car net))
        (extract-right-subnet
         (list (make-point net-name () () ()
                           '(right first ring)))
         (append
          (list
           (append1 subnet (make-point net-name () () ()
                                       '(bottom last ring)))
           (list (make-point net-name () () ()
                             '(left first ring))
                 (make-point net-name () () ()
                             '(left last ring)))
           (list (make-point net-name () () ()
                             '(top first ring))
                 (make-point net-name () () ()
                             '(top last ring))))
           subnets)
         net
         net-name))
       ((is-point-bottom? (car net))
        (extract-bottom-subnet
         (append1 subnet (car net)) subnets (cdr net) net-name))
  ((is-point-left? (car net))
        (extract-left-subnet
         (list (make-point net-name () () ()
                           '(left first ring)))
         (append
          (list
           (append1 subnet
                    (make-point net-name () () ()
                                '(bottom last ring))))
           subnets)
         net
         net-name)))))

(def extract-left-subnet
 (lambda (subnet subnets net net-name)
  (cond ((null net) (cons subnet subnets))
```

```
        ((is-point-top? (car net))
         (extract-top-subnet
          (list (make-point net-name () () () '(top first ring)))
     (append
           (list (append1 subnet
                          (make-point net-name () () ()
                                      '(left last ring))))
           subnets)
          net
          net-name))
        ((is-point-right? (car net))
         (extract-right-subnet
          (list (make-point net-name () () ()
                            '(right first ring)))
          (append
           (list (append1 subnet
                          (make-point net-name () () ()
                                      '(left last ring)))
                 (list (make-point net-name () () ()
                                   '(top first ring))
                       (make-point net-name () () ()
                                   '(top last ring))))
           subnets)
          net
          net-name))
        ((is-point-bottom? (car net))
         (extract-bottom-subnet
          (list (make-point net-name () () ()
                            '(bottom first ring)))
          (append
           (list (append1 subnet
                          (make-point net-name () () ()
                                      '(left last ring)))
                 (list (make-point net-name () () ()
                                   '(top first ring))
                       (make-point net-name () () ()
                                   '(top last ring)))
                 (list (make-point net-name () () ()
                                   '(right first ring))
                       (make-point net-name () () ()
                                   '(right last ring))))
           subnets)
          net
          net-name))
        ((is-point-left? (car net))
         (extract-left-subnet
          (append1 subnet (car net))
          subnets (cdr net) net-name)))))

(declare (special item))

(def extract-basic-nets
 (lambda (item)
  (alpha (lambda (name)
```

```
          (such-that (find-all item name)
                     (lambda (point)
                      (member? 'ring (point-attributes point)))))
    (setify (alpha (function point-name)
                   (find-attributes item '(ring)))))))


(declare (unspecial item))

(def order-basic-nets
 (lambda (basic-nets)
  (alpha (lambda (basic-net)
          (sort basic-net
                (function basic-net-point-further-left?)))
          basic-nets)))

(declare (special right top))

(def rotate-basic-nets
 (lambda (basic-nets right top)
  (alpha (lambda (basic-net)
          (rotate basic-net
                  (rotation-amount basic-net right top)))
         basic-nets)))

(declare (unspecial right top))

(def rotation-amount
 (lambda (basic-net right top)
  (cond ((or (null basic-net) (= (length basic-net) 1)) 0)
        (t (rotation-count
             (rotation-amount1 basic-net
                               (car (last basic-net)) right top))))))

(def rotation-amount1
 (lambda (basic-net last right top)
  (let ((head (make-rotation
               0
               (basic-net-distance last
                                   (car basic-net)
                                   right top))))
   (cond
    ((= (length basic-net) 1) head)
    (t (let ((tail (rotation-amount1 (cdr basic-net)
                                     (car basic-net) right top)))
   (cond ((> (rotation-distance tail)
             (rotation-distance head))
          (make-rotation (1+ (rotation-count tail))
                         (rotation-distance tail)))
          (t head)))))))))

(def basic-net-left-point
 (lambda (basic-net)
  (car basic-net)))
```

```
(def basic-net-right-point
 (lambda (basic-net)
  (car (last basic-net))))

(def basic-net-point-further-left?
 (lambda (point1 point2)
  (cond ((is-point-first? point1) (not (is-point-first? point2)))
   ((is-point-last? point1) ())
        ((is-point-first? point2) ())
        ((is-point-last? point2) t)
        ((is-point-top? point1)
         (< (point-x point1) (point-x point2)))
        ((is-point-right? point1)
         (> (point-y point1) (point-y point2)))
        ((is-point-bottom? point1)
         (> (point-x point1) (point-x point2)))
        ((is-point-left? point1)
         (< (point-y point1) (point-y point2))))))

(def basic-net-overlap?
 (lambda (basic-net left-point)
  (let ((right-point (basic-net-right-point basic-net)))
   (cond ((is-point-first? right-point)
          (is-point-first? left-point))
         ((is-point-last? right-point) t)
         ((is-point-first? left-point) t)
         ((is-point-last? left-point) ())
         ((is-point-top? right-point)
          (> (point-x right-point) (- (point-x left-point) 7)))
  ((is-point-right? right-point)
          (< (point-y right-point) (+ (point-y left-point) 7)))
  ((is-point-bottom? right-point)
          (< (point-x right-point) (+ (point-x left-point) 7)))
  ((is-point-left? right-point)
          (> (point-y right-point)
             (- (point-y left-point) 7)))))))

(def basic-net-distance
 (lambda (point1 point2 right top)
  (let ((x1 (point-x point1))
        (y1 (point-y point1))
        (x2 (point-x point2))
        (y2 (point-y point2)))
   (cond
    ((is-point-top? point1)
     (cond
      ((is-point-top? point2)
       (cond
        ((< x1 x2) (- x2 x1))
        (t (- (+ right right top top) (- x1 x2)))))
      ((is-point-right? point2) (+ (- right x1) (- top y2)))
      ((is-point-bottom? point2)
       (+ (- right x1) top (- right x2)))
```

```
      ((is-point-left? point2) (+ (- right x1) top right y2))))
      ((is-point-right? point1)
       (cond ((is-point-top? point2) (+ y1 right top x2))
             ((is-point-right? point2)
        (cond
         ((> y1 y2) (- y1 y2))
         (t (- (+ right right top top) (- y2 y1)))))
        ((is-point-bottom? point2) (+ y1 (- right x2)))
        ((is-point-left? point2) (+ y1 right y2))))
        ((is-point-bottom? point1)
         (cond
          ((is-point-top? point2) (+ x1 top x2))
          ((is-point-right? point2) (+ x1 top right (- top y2)))
          ((is-point-bottom? point2)
          (cond
           ((> x1 x2) (- x1 x2))
           (t (- (+ right right top top) (- x2 x1)))))
           ((is-point-left? point2) (+ x1 y1))))
           ((is-point-left? point1)
            (cond
             ((is-point-top? point2) (+ (- top y1) x2))
             ((is-point-right? point2)
              (+ (- top y1) right (- top y2)))
             ((is-point-bottom? point2)
              (+ (- top y1) right top (- right x2)))
             ((is-point-left? point2)
              (cond
               ((< y1 y2) (- y2 y1))
               (t (- (+ right right top top) (- y2 y1)))))))))))

(def point-side
 (lambda (point)
  (cond ((is-point-top? point) 'top)
        ((is-point-right? point) 'right)
        ((is-point-bottom? point) 'bottom)
        ((is-point-left? point) 'left))))

(def point-value
 (lambda (point)
  (cond ((is-point-top? point) (point-x point))
        ((is-point-right? point) (point-y point))
        ((is-point-bottom? point) (point-x point))
        ((is-point-left? point) (point-y point)))))

(def is-point-inside?
 (lambda (point)
  (member? 'inside (point-attributes point))))

(def is-point-outside?
 (lambda (point)
  (member? 'outside (point-attributes point))))

(def is-point-top?
 (lambda (point)
```

```
  (member? 'top (point-attributes point))))

(def is-point-bottom?
 (lambda (point)
  (member? 'bottom (point-attributes point))))

(def is-point-left?
 (lambda (point)
  (member? 'left (point-attributes point))))

(def is-point-right?
 (lambda (point)
  (member? 'right (point-attributes point))))

(def is-point-first?
 (lambda (point)
  (member? 'first (point-attributes point))))

(def is-point-last?
 (lambda (point)
  (member? 'last (point-attributes point))))

;;;layout pins

(def pins-conductivity
 (lambda (pins)
  (slash-alpha
   pins
   0.0
   (function plus)
   (lambda (pin) (pad-conductivity (pin-pad pin))))))

(defsymbol layout-pins
 (pins power intended-right intended-top ring-width logo)
 (let ((pins-power
        (conductivity-to-power-bus-width
         (pins-conductivity pins)
         (pad-class-default-power-bus-width))))
  (let ((dimensions
         (pins-dimensions pins pins-power ring-width
                          intended-right intended-top)))
   (let ((top (dimensions-top dimensions))
         (right (dimensions-right dimensions))
         (bottom (dimensions-bottom dimensions))
         (left (dimensions-left dimensions)))
    (let ((pins-layout (place-pins pins dimensions pins-power)))
     (let ((power-point (find pins-layout '(power)))
           (ground-point (find pins-layout '(ground))))
      (merge (cond ((member? 'logo option-list)
                    (move (first-quadrant
                             (title logo 'NM 'nonie.r.10))
                          (+ left pins-power 3)
                          (+ bottom pins-power 3)))
                   (t (null-item)))
```

```
            pins-layout
            (layout-power-ring pins-power power dimensions
                               intended-right intended-top
                               power-point ground-point)
            (layout-ground-ring pins-power power dimensions
                                intended-right intended-top
                                power-point ground-point)))))))

(def layout-power-ring
 (lambda (pins-power power dimensions intended-right intended-top
    power-point ground-point)
  (let ((top (dimensions-top dimensions))
        (right (dimensions-right dimensions))
        (bottom (dimensions-bottom dimensions))
        (left (dimensions-left dimensions)))
   (merge (rect 'NM left
                (- top pins-power)
                right
                top)
          (rect 'NM (- right pins-power)
                bottom
                right
                top)
          (rect 'NM left
                bottom
                right
                (+ bottom pins-power))
          (rect 'NM left
                bottom
                (+ left pins-power)
                top)
          (cond ((is-point-top? power-point)
                 (warning "Power pin can not be on top
                          side of circuit")
                 (rect 'NM (- (point-x power-point)
                              (/up power 2))
                           intended-top
                           (+ (point-x power-point)
                              (/up power 2))
                           (point-y power-point)))
                ((is-point-right? power-point)
                 (rect 'NM intended-right
                           (- (point-y power-point)
                              (/up power 2))
                           (point-x power-point)
                           (+ (point-y power-point)
                              (/up power 2))))
                ((is-point-bottom? power-point)
                 (rect 'NM (- (point-x power-point)
                              (/up power 2))
                           (point-y power-point)
                           (+ (point-x power-point)
                              (/up power 2))
                           0))
```

```
                ((is-point-left? power-point)
                 (rect 'NM (point-x power-point)
                           (- (point-y power-point)
                              (/up power 2))
                           0
                           (+ (point-y power-point)
                              (/up power 2)))))))))

(def layout-ground-ring
 (lambda (pins-power power dimensions intended-right intended-top
     power-point ground-point)
  (let ((offset (+ pins-power (pad-class-basic-height))))
   (let ((top (- (dimensions-top dimensions) offset))
         (right (- (dimensions-right dimensions) offset))
         (bottom (+ (dimensions-bottom dimensions) offset))
         (left (+ (dimensions-left dimensions) offset)))
    (merge
     (cond ((is-point-top? ground-point)
            (rect 'NM (- (point-x ground-point) (/up power 2))
                  intended-top
                       (+ (point-x ground-point) (/up power 2))
                  (point-y ground-point)))
           ((is-point-right? ground-point)
            (warning "Ground pin must be on top side of circuit")
      (rect 'NM intended-right
                       (- (point-y ground-point) (/up power 2))
                  (point-x ground-point)
                       (+ (point-y ground-point) (/up power 2))))
           ((is-point-bottom? ground-point)
            (rect 'NM (- (point-x ground-point) (/up power 2))
                  (point-y ground-point)
                       (+ (point-x ground-point) (/up power 2))
                  0))
           ((is-point-left? ground-point)
            (rect 'NM (point-x ground-point)
                       (- (point-y ground-point) (/up power 2))
                  0
                       (+ (point-y ground-point) (/up power 2)))))
     (cond ((is-point-top? power-point)
            (merge
             (rect 'NM left
                       (- top pins-power)
                       (- (point-x power-point)
                          (/up (pad-class-width) 2))
                       top)
              (rect 'NM (+ (point-x power-point)
                          (/up (pad-class-width) 2))
                       (- top pins-power)
                       right
                       top)
              (rect 'NM (- right pins-power)
                       bottom
                       right
                       top)
```

```
 (rect 'NM left
          bottom
          right
          (+ bottom pins-power))
 (rect 'NM left
          bottom
          (+ left pins-power)
          top)))
((is-point-right? power-point)
(merge
 (rect 'NM left
          (- top pins-power)
          right
          top)
 (rect 'NM (- right pins-power)
          (+ (point-y power-point)
             (/up (pad-class-width) 2))
          right
          top)
 (rect 'NM (- right pins-power)
          bottom
          right
          (- (point-y power-point)
             (/up (pad-class-width) 2)))
 (rect 'NM left
          bottom
          right
          (+ bottom pins-power))
 (rect 'NM left
          bottom
          (+ left pins-power)
          top)))
((is-point-bottom? power-point)
(merge
 (rect 'NM left
          (- top pins-power)
          right
          top)
 (rect 'NM (- right pins-power)
          bottom
          right
          top)
 (rect 'NM (+ (point-x power-point)
             (/up (pad-class-width) 2))
          bottom
          right
          (+ bottom pins-power))
 (rect 'NM left
          bottom
          (- (point-x power-point)
             (/up (pad-class-width) 2))
          (+ bottom pins-power))
 (rect 'NM left
          bottom
```

```
                    (+ left pins-power)
                    top)))
            ((is-point-left? power-point)
            (merge
             (rect 'NM left
                    (- top pins-power)
                    right
                    top)
             (rect 'NM (- right pins-power)
                    bottom
                    right
                    top)
             (rect 'NM left
                    bottom
                    right
                    (+ bottom pins-power))
             (rect 'NM left
                    bottom
                    (+ left pins-power)
                    (- (point-y power-point)
                       (/up (pad-class-width) 2)))
             (rect 'NM left
                    (+ (point-y power-point)
                       (/up (pad-class-width) 2))
                    (+ left pins-power)
                    top)))))))))

(def extend-right
 (lambda (pins intended-right)
  (let ((maximum-number-pins-horizontally
          (/ intended-right (pad-class-width)))
        (number-pins-per-side
          (/up (slash-alpha pins 0 (function max)
                            (function pin-pin-number))
               3)))
   (cond
    ((<= number-pins-per-side maximum-number-pins-horizontally)
     intended-right)
    (t (* (pad-class-width) number-pins-per-side))))))

(def extend-top
 (lambda (pins intended-top)
  (let ((maximum-number-pins-vertically
          (/ intended-top (pad-class-width)))
        (number-pins-per-side
         (/up (slash-alpha pins 0 (function max)
                           (function pin-pin-number))
              3)))
   (cond
    ((<= number-pins-per-side maximum-number-pins-vertically)
     intended-top)
    (t (* (pad-class-width) number-pins-per-side))))))
```

```
(def pins-dimensions
 (lambda (pins pins-power ring-width intended-right intended-top)
(let ((maximum-number-pins-horizontally
           (/ intended-right (pad-class-width)))
        (maximum-number-pins-vertically
           (/ intended-top (pad-class-width)))
        (number-pins-per-side
           (/up (slash-alpha pins 0 (function max)
                             (function pin-pin-number))
                3)))
   (cond
    ((and (<= number-pins-per-side
              maximum-number-pins-horizontally)
          (<= number-pins-per-side
              maximum-number-pins-vertically))
     (make-dimensions
      number-pins-per-side
      (+ intended-top (ring-width-top ring-width)
         (side-extension 'top pins number-pins-per-side
                         pins-power))
      (+ intended-right (ring-width-right ring-width)
         (side-extension 'right pins number-pins-per-side
                         pins-power))
      (- 0 (ring-width-bottom ring-width)
         (side-extension 'bottom pins number-pins-per-side
                         pins-power))
      (- 0 (ring-width-left ring-width)
         (side-extension 'left pins number-pins-per-side
                         pins-power))))
    ((<= number-pins-per-side maximum-number-pins-horizontally)
(make-dimensions
      number-pins-per-side
      (+ (* (pad-class-width) number-pins-per-side)
         (ring-width-top ring-width)
         (side-extension 'top pins number-pins-per-side
                         pins-power))
      (+ intended-right (ring-width-right ring-width)
         (side-extension 'right pins number-pins-per-side
                         pins-power))
      (- 0 (ring-width-bottom ring-width)
         (side-extension 'bottom pins number-pins-per-side
                         pins-power))
      (- 0 (ring-width-left ring-width)
         (side-extension 'left pins number-pins-per-side
                         pins-power))))
    ((<= number-pins-per-side maximum-number-pins-vertically)
(make-dimensions
      number-pins-per-side
      (+ intended-top (ring-width-top ring-width)
         (side-extension 'top pins number-pins-per-side
                         pins-power))
      (+ (* (pad-class-width) number-pins-per-side)
         (ring-width-right ring-width)
         (side-extension 'right pins number-pins-per-side
```

```
                        pins-power))
      (- 0 (ring-width-bottom ring-width)
         (side-extension 'bottom pins number-pins-per-side
                        pins-power))
      (- 0 (ring-width-left ring-width)
         (side-extension 'left pins number-pins-per-side
                        pins-power))))
    (t
     (make-dimensions
      number-pins-per-side
      (+ (* (pad-class-width) number-pins-per-side)
         (ring-width-top ring-width)
         (side-extension 'top pins number-pins-per-side
                        pins-power))
      (+ (* (pad-class-width) number-pins-per-side)
         (ring-width-right ring-width)
         (side-extension 'right pins number-pins-per-side
                        pins-power))
      (- 0
         (ring-width-bottom ring-width)
         (side-extension 'bottom pins number-pins-per-side
                   pins-power))
      (- 0
         (ring-width-left ring-width)
         (side-extension 'left pins number-pins-per-side
                        pins-power)))))))

(def pin-height
 (lambda (pin power)
  (+ power
     (pad-class-basic-height)
     power
     (pad-basic-extension (pin-pad pin)))))


(declare (special side number-pins-per-side power))

(def side-extension
 (lambda (side pins number-pins-per-side power)
  (slash-alpha
   (such-that
    pins
    (lambda (pin)
     (cond
      ((eq side 'top)
       (<= (pin-pin-number pin) number-pins-per-side))
 ((eq side 'right)
       (and (> (pin-pin-number pin) number-pins-per-side)
       (<= (pin-pin-number pin)
                (* 2 number-pins-per-side))))
            ((eq side 'bottom)
             (and (> (pin-pin-number pin)
                     (* 2 number-pins-per-side))
                  (<= (pin-pin-number pin)
```

```
                    (* 3 number-pins-per-side))))
          ((eq side 'left) (> (pin-pin-number pin)
                              (* 3 number-pins-per-side))))))
    (+ power (pad-class-basic-height) power 3)
    (function max)
    (lambda (pin) (pin-height pin power)))))

(declare (unspecial side number-pins-per-side power))

(def place-pins
 (lambda (pins dimensions power)
  (cond ((null pins) (null-item))
        (t (merge (place-pin (car pins) dimensions power)
                  (place-pins (cdr pins) dimensions power))))))

(def place-pin
 (lambda (pin dimensions power)
  (let ((number-pins-per-side
          (dimensions-number-pins-per-side dimensions))
        (top (dimensions-top dimensions))
        (right (dimensions-right dimensions))
        (bottom (dimensions-bottom dimensions))
        (left (dimensions-left dimensions))
        (pin-number (pin-pin-number pin)))
   (cond ((<= pin-number number-pins-per-side)
          (move (mirrorx (layout-pad (pin-pad pin) power 'top))
         (* (pad-class-width) (1- pin-number))
                top))
         ((<= pin-number (* 2 number-pins-per-side))
          (move (rotccw (layout-pad (pin-pad pin) power 'right))
                right
                (* (pad-class-width)
                   (- (* 2 number-pins-per-side) pin-number))))
         ((<= pin-number (* 3 number-pins-per-side))
          (move (layout-pad (pin-pad pin) power 'bottom)
                (* (pad-class-width)
                   (- (* 3 number-pins-per-side) pin-number))
                bottom))
         ((<= pin-number (* 4 number-pins-per-side))
          (move (rotccw (mirrorx (layout-pad (pin-pad pin)
                                             power 'left)))
                left
                (* (pad-class-width)
                   (- pin-number
                      (* 3 number-pins-per-side) 1)))))))))

(def /up
 (lambda (x y)
  (cond ((equal x (times y (fix (quotient x y))))
         (fix (quotient x y)))
        (t (1+ (fix (quotient x y)))))))

;;;the following routines must be changed when adding new pad
classes
```

```
(def pad-class
 ;;  16 Apr 87 J Harmon made pad20b the default pad file for all
 ;;  minimum feature sizes except 250 centimicrons.
 (lambda ()
  (cond ((= (minimum-feature-size) 250) 'rinout)
        (t 'pad20b))))

(def pad-class-default-power-bus-width
 (lambda ()
  (cond ((eq (pad-class) 'rinout) 16)
        ((eq (pad-class) 'pad20b) 8))))

(def pad-class-basic-height
 (lambda ()
  (cond ((eq (pad-class) 'rinout) 82)
        ((eq (pad-class) 'pad20b) 112))))

(def pad-class-width
 (lambda ()
  (cond ((eq (pad-class) 'rinout) 100)
        ((eq (pad-class) 'pad20b) 128))))

(def pad-conductivity
 (lambda (pad)
  (cond ((eq (pad-class) 'rinout)
         (cond ((is-pad-blank? pad) 0.0)
               ((is-pad-input? pad) 1.66666)
               ((is-pad-output4? pad) 0.31250)
               ((is-pad-output8? pad) 0.31250)
               ((is-pad-tri-state4? pad) 0.15625)
               ((is-pad-tri-state8? pad) 0.15625)
               ((is-pad-i/o4? pad) 0.14285)
               ((is-pad-i/o8? pad) 0.14285)
               ((is-pad-power? pad) 0.0)
               ((is-pad-ground? pad) 0.0)
               ((is-pad-phia? pad) 1.66666)
               ((is-pad-phib? pad) 1.66666)
               ((is-pad-phic? pad) 1.66666)))
        ((eq (pad-class) 'pad20b)
         (cond ((is-pad-blank? pad) 0.0)
               ((is-pad-input? pad) 1.66666)
               ((is-pad-output4? pad) 0.31250)
               ((is-pad-output8? pad) 0.31250)
               ((is-pad-tri-state4? pad) 0.15625)
               ((is-pad-tri-state8? pad) 0.15625)
               ((is-pad-i/o4? pad) 0.14285)
               ((is-pad-i/o8? pad) 0.14285)
               ((is-pad-power? pad) 0.0)
               ((is-pad-ground? pad) 0.0)
               ((is-pad-phia? pad) 1.66666)
               ((is-pad-phib? pad) 1.66666)
               ((is-pad-phic? pad) 1.66666))))))
```

```
(def pad-basic-extension
 (lambda (pad)
  (cond ((eq (pad-class) 'rinout)
         (cond ((is-pad-blank? pad) 3)
               ((is-pad-input? pad) 29)
               ((is-pad-output4? pad) 42)
               ((is-pad-output8? pad) 42)
               ((is-pad-tri-state4? pad) 67)
               ((is-pad-tri-state8? pad) 67)
               ((is-pad-i/o4? pad) 96)
               ((is-pad-i/o8? pad) 96)
               ((is-pad-power? pad) 3)
               ((is-pad-ground? pad) 3)
               ((is-pad-phia? pad) 29)
               ((is-pad-phib? pad) 29)
               ((is-pad-phic? pad) 29)))
        ((eq (pad-class) 'pad20b)
         (cond ((is-pad-blank? pad) 3)
               ((is-pad-input? pad) 21)
               ((is-pad-output4? pad) 42)
               ((is-pad-output8? pad) 42)
               ((is-pad-tri-state4? pad) 67)
               ((is-pad-tri-state8? pad) 67)
               ((is-pad-i/o4? pad) 74)
               ((is-pad-i/o8? pad) 74)
               ((is-pad-power? pad) 3)
               ((is-pad-ground? pad) 3)
               ((is-pad-phia? pad) 21)
               ((is-pad-phib? pad) 21)
               ((is-pad-phic? pad) 21))))))

(def pad-transistor-count
 (lambda (pad)
  (cond ((eq (pad-class) 'rinout)
         (cond ((is-pad-blank? pad) '(0 0))
               ((is-pad-input? pad) '(4 3))
               ((is-pad-output4? pad) '(6 4))
               ((is-pad-output8? pad) '(6 4))
               ((is-pad-tri-state4? pad) '(11 5))
               ((is-pad-tri-state8? pad) '(11 5))
               ((is-pad-i/o4? pad) '(14 8))
               ((is-pad-i/o8? pad) '(14 8))
               ((is-pad-power? pad) '(0 0))
               ((is-pad-ground? pad) '(0 0))
               ((is-pad-phia? pad) '(4 3))
               ((is-pad-phib? pad) '(4 3))
               ((is-pad-phic? pad) '(4 3))))
        ((eq (pad-class) 'pad20b)
         (cond ((is-pad-blank? pad) '(0 0))
               ((is-pad-input? pad) '(4 3))
               ((is-pad-output4? pad) '(6 4))
               ((is-pad-output8? pad) '(6 4))
               ((is-pad-tri-state4? pad) '(11 5))
               ((is-pad-tri-state8? pad) '(11 5))
```

```
                ((is-pad-i/o4? pad) '(14 8))
                ((is-pad-i/o8? pad) '(14 8))
                ((is-pad-power? pad) '(0 0))
                ((is-pad-ground? pad) '(0 0))
                ((is-pad-phia? pad) '(4 3))
                ((is-pad-phib? pad) '(4 3))
                ((is-pad-phic? pad) '(4 3))))))

(def layout-pad
 (lambda (pad power side)
  (cond
   ((eq (pad-class) 'rinout)
    (cond
     ((is-pad-blank? pad) (null-item))
     ((is-pad-input? pad)
      (layout-rinout-input-pad
       power (input-pad-name pad) (input-pad-in-wire pad) side))
     ((is-pad-output4? pad)
      (layout-rinout-output4-pad
       power (output4-pad-name pad) (output4-pad-out-wire pad) side))
     ((is-pad-output8? pad)
      (layout-rinout-output8-pad
       power (output8-pad-name pad) (output8-pad-out-wire pad) side))
     ((is-pad-tri-state4? pad)
      (layout-rinout-tri-state4-pad
         power (tri-state4-pad-name pad)
         (tri-state4-pad-out-wire pad)
         (tri-state4-pad-drive-wire pad) side))
     ((is-pad-tri-state8? pad)
      (layout-rinout-tri-state8-pad
         power (tri-state8-pad-name pad)
         (tri-state8-pad-out-wire pad)
         (tri-state8-pad-drive-wire pad) side))
     ((is-pad-i/o4? pad)
      (layout-rinout-i/o4-pad power (i/o4-pad-name pad)
                              (i/o4-pad-in-wire pad)
                              (i/o4-pad-out-wire pad)
                              (i/o4-pad-drive-wire pad) side))
     ((is-pad-i/o8? pad)
      (layout-rinout-i/o8-pad power (i/o8-pad-name pad)
                              (i/o8-pad-in-wire pad)
                              (i/o8-pad-out-wire pad)
                              (i/o8-pad-drive-wire pad) side))
     ((is-pad-power? pad)
      (layout-rinout-power-pad power side))
     ((is-pad-ground? pad)
      (layout-rinout-ground-pad power side))
     ((is-pad-phia? pad)
      (layout-rinout-phia-pad power (make-phia-wire) side))
     ((is-pad-phib? pad)
      (layout-rinout-phib-pad power (make-phib-wire) side))
     ((is-pad-phic? pad)
      (layout-rinout-phic-pad power (make-phic-wire) side))))
     ((eq (pad-class) 'pad20b)
```

```
(cond
 ((is-pad-blank? pad) (null-item))
 ((is-pad-input? pad)
  (layout-pad20b-input-pad
   power (input-pad-name pad) (input-pad-in-wire pad) side))
 ((is-pad-output4? pad)
  (layout-pad20b-output4-pad
   power (output4-pad-name pad)
   (output4-pad-out-wire pad) side))
 ((is-pad-output8? pad)
  (layout-pad20b-output8-pad
   power (output8-pad-name pad) (output8-pad-out-wire pad) side))
 ((is-pad-tri-state4? pad)
  (layout-pad20b-tri-state4-pad
    power
    (tri-state4-pad-name pad)
    (tri-state4-pad-out-wire pad)
    (tri-state4-pad-drive-wire pad) side))
 ((is-pad-tri-state8? pad)
  (layout-pad20b-tri-state8-pad
    power (tri-state8-pad-name pad)
    (tri-state8-pad-out-wire pad)
    (tri-state8-pad-drive-wire pad) side))
 ((is-pad-i/o4? pad)
  (layout-pad20b-i/o4-pad power (i/o4-pad-name pad)
                          (i/o4-pad-in-wire pad)
                          (i/o4-pad-out-wire pad)
                          (i/o4-pad-drive-wire pad) side))
((is-pad-i/o8? pad)
  (layout-pad20b-i/o8-pad power (i/o8-pad-name pad)
                          (i/o8-pad-in-wire pad)
                          (i/o8-pad-out-wire pad)
                          (i/o8-pad-drive-wire pad) side))
((is-pad-power? pad)
  (layout-pad20b-power-pad power side))
((is-pad-ground? pad)
  (layout-pad20b-ground-pad power side))
((is-pad-phia? pad)
 (layout-pad20b-phia-pad power (make-phia-wire) side))
((is-pad-phib? pad)
 (layout-pad20b-phib-pad power (make-phib-wire) side))
((is-pad-phic? pad)
 (layout-pad20b-phic-pad power (make-phic-wire) side))))))
```

APPENDIX B

MONTEREY FUNCTIONS

This Appendix contains all the new functions, as well as, altered MacPitts' functions that play a role in the new pad router and pad placement process. Comments are identified by a leading semicolon and capital letters. They pertain to the code immediately after the comment.

```
      ; FUNCTION RESPONSIBLE FOR ASSIGNING DATA-PATH TERMINALS
      ; TO CONNECT TO PADS THROUGH EITHER THE LEFT OR RIGHT
      ; SIDES.  THE ORIGINAL FUNCTION LACKED THE CONDITIONAL
      ; STATEMENT.  ALL POINTS WERE SENT THROUGH THE LEFT SIDE
      ; BY MAKE-LEFT-TIP
(def get-basic-buses-from-port-output-unit
 (lambda (number-of-units tail unit unit-number)
  (update-basic-buses
   (update-basic-buses
    tail
    (make-port-output-id (port-output-unit-name unit))
    (cond
     ((> unit-number (/ number-of-units 2)) (make-right-tip))
     (t (make-left-tip))))
   (make-port-output-id (port-output-unit-name unit))
   (make-output-tip unit-number))))


; TOP LEVEL OF LAYOUT ROUTINES

(declare (special gates top-part))

(def layout-object
 (lambda (object)
  (prog (definitions flags data-path control pins gates straps
         conductivity power data-path-length control-length
         flags-length top-width bottom-width data-path-layout
         control-layout flags-layout river-layout wing-layout
         skeleton-layout internal-layout pins-layout ring-layout
         layout nets ring-width top-part bottom-part top-bank
         bottom-bank river-width bottom-part-river-points
         intended-right intended-top extended-right extended-top)
  (setq definitions (object-definitions object))
  (setq flags (object-flags object))
  (setq data-path (object-data-path object))
```

```
  (setq control (object-control object))
  (setq pins (object-pins object))
(herald "Extruding gates")
  (setq gates (extrude-gates control flags))
  (statistic (concat "Control has " (length gates) " columns"))
   (cond ((member? 'opt-c option-list)
          (setq gates
           (nthelem-list
            (order (extrude-basic-straps gates)
                   gates
                   (count (length gates))
                   (function junction-gate-number)
                   (flambda (basic-strap) basic-strap)
                   (flambda (gate1 gate2)
                            (gate-before? gate1 gate2 gates))
              (flambda (gate1 gate2)
                            (gate-after? gate1 gate2 gates)))
          gates))))
  (setq gates (insert-nor-ground-lines gates))
(herald "Extruding straps")
  (setq straps (extrude-straps gates))
  (statistic
    (concat "Circuit has "
            (slash-alpha
             (list (flags-transistor-count flags)
                   (data-path-transistor-count data-path
                                               definitions)
             (control-transistor-count gates straps)
      (pins-transistor-count pins))
             0
             (function +)
             (flambda (x) (+ (car x) (cadr x))))
            " transistors"))
   (statistic
    (concat "Control has "
            (slash-alpha straps 0 (function max)
                                  (function strap-track-number))
            " tracks"))
  (setq conductivity (plus (data-path-conductivity data-path
                                                   definitions)
                           (control-conductivity gates straps)
                          (flags-conductivity flags)))
  (setq power (conductivity-to-power-bus-width conductivity 11))
  (statistic (concat "Power consumption is "
                     (conductivity-to-power-consumption
                      (plus conductivity
                            (pins-conductivity pins)))
                     " Watts"))
   (setq data-path-length
         (max (data-path-required-length data-path definitions)
           4))
   (setq control-length (control-required-length gates))
   (setq flags-length (max (flags-required-length flags power)
                     4))
```

```
   (setq top-width (max (data-path-required-width
                          data-path power definitions)
                        (flags-required-width flags power)))
(setq bottom-width (control-required-width straps))
(herald "Laying out data-path")
   (setq data-path-layout
         (layout-data-path data-path power
                           top-width definitions))
(herald "Laying out control")
   (setq control-layout (layout-control gates straps
                                        power bottom-width))
(herald "Laying out flags")
   (setq flags-layout (layout-flags flags power top-width))
(herald "Laying out river")
   (setq top-part
    (merge (move data-path-layout (+ power 3) 0)
           (move flags-layout (+ power 3 data-path-length
                          3 power 3) 0)))
   (setq bottom-part
         (move control-layout (+ power 3) (- power 4)))
   (setq bottom-part-river-points
         (find-attributes bottom-part '(river)))
   (setq top-bank
         (sort (alpha (flambda
                        (point)
                        (point-x (find top-part
                                       (point-name point))))
                 bottom-part-river-points)
                (function <)))
   (setq bottom-bank
         (sort (alpha (function point-x)
                      bottom-part-river-points)
               (function <)))
   (setq river-width
         (+ (river-span 'NP 2 top-bank bottom-bank)
            (wing-span bottom-part)
            (- 4 power)))
   (setq intended-top
         (+ power bottom-width power river-width (driver-width)
          power top-width power 3 power))
   (setq intended-right
         (+ power 3 (max control-length
                         (+ data-path-length 3
                            power 3 flags-length))
            3 power))
   (setq river-layout
    (river 'NP 2 (wing-span bottom-part) top-bank bottom-bank))
 (herald "Laying out wing")
   (setq wing-layout
         (layout-wing (sort (find-attributes bottom-part '(wing))
                            (flambda (point1 point2)
                                     (< (point-x point1)
                                     (point-x point2))))))
(herald "Laying out skeleton")
```

```
   (setq skeleton-layout
         (layout-skeleton power intended-top intended-right
                     data-path-length bottom-width
                          river-width))
   (setq internal-layout
         (merge
          (move top-part 0
                        (+ power bottom-width power river-width
                           (driver-width) power))
          bottom-part
          (move (rotcw river-layout) 0
                                     (+ power bottom-width power
                                        river-width))
          (move wing-layout 0
                           (+ power bottom-width 4))
           skeleton-layout))
   (herald "Laying out pins")
   (setq pins-layout
         (layout-pins pins
                      top-part
                      bottom-part
                      power
                      intended-right
                      intended-top
                      (make-ring-width 0 0 0 0)
                      (lookup-logo definitions)))

     ; IF NOT SUFFICIENT SPACE TO ACCOMMODATE NUMBER OF PADS;
     ; SPECIFIED BY NUMBER-PADS-PER-SIDE, EXTEND HORIZONTAL
     ; DIMENSION UNTIL THEY FIT.
   (setq extended-right (extend-right pins intended-right
                                      intended-top))

     ; IF NOT SUFFICIENT SPACE TO ACCOMMODATE NUMBER OF PADS;
     ; SPECIFIED BY NUMBER-PADS-PER-SIDE, EXTEND VERTICAL
     ; DIMENSION UNTIL THEY FIT.
   (setq extended-top (extend-top pins intended-top extended-
right))

     ; CALCULATES CHANNEL WIDTHS FOR EACH OF THE FOUR SEGMENTS
     ; OF THE PAD ROUTER ROUTING REGIONS
   (setq ring-width
         (get-ring-width (merge internal-layout pins-layout)
                     extended-right extended-top))

     ; SECOND PASS THRU LAYOUT-PINS DIFFERS FROM FIRST IN THAT
     ; CORRECT RING-WIDTHS ARE AVAILABLE
   (setq pins-layout
         (layout-pins pins
                      top-part
                      bottom-part
                      power
                      intended-right
                      intended-top
```

```
                    ring-width
                    (lookup-logo definitions)))


  ; EXTRACT NET-LISTS FOR NETS THAT CONNECT TO LEFT SIDE
  ; OF CIRCUIT
(setq left-ring-nets
      (append (list (sort-y
                     (get-nets
                      (append
                       '(0 0 0 0)
                       (list (mapcar
                              'car
                              (extract-basic-nets
                                 internal-layout))))
                      '(left)) ()))
              (list (prep-pad-bank pins-layout 'left))))

  ; EXTRACT NET-LISTS FOR PADS THAT CONNECT TO RIGHT SIDE
  ; OF CIRCUIT.
(setq right-ring-nets
      (append (list (sort-y
                     (get-nets
                      (append
                       '(0 0 0 0)
                       (list (mapcar
                              'car
                              (extract-basic-nets
                                 internal-layout))))
                      '(right)) ()))
              (list (prep-pad-bank pins-layout 'right))))

  ; PRODUCES LAYOUT OF NETS BETWEEN CIRCUIT AND PADS.
(setq ring-layout
      (moat left-ring-nets
            right-ring-nets
            'NM
            4
            ring-width))

(setq layout
      (first-quadrant (merge internal-layout pins-layout
                             ring-layout)))
(statistic (concat "Dimensions are "
  ;;jh replaced minimum-feature-size with lambda-spacing.
               (quotient (times (right layout)
                                (lambda-spacing))
                         100000.0)
                    " mm by "
                    (quotient (times (top layout)
                                     (lambda -spacing))
                              100000.0)
                    " mm"))
(return layout))))
```

```
(declare (unspecial gates top-part))

      ; TOP LEVEL OF PAD LAYOUT ROUTINES
(defsymbol layout-pins (pins top-part bottom-part power
                             intended-right intended-top
                             ring-width logo)
 (let ((extended-right (extend-right pins intended-right
                                     intended-top)))
  (let ((extended-top (extend-top pins intended-top
                                  extended-right)))

       ; CALCULATE WIDTH OF PAD POWER AND GROUND SUPPLY RAILS.
 (let ((pins-power (conductivity-to-power-bus-width
                       (pins-conductivity pins)
                       (pad-class-default-power-bus-width))))

       ; DEVELOPS LIST THAT SPECIFY PAD LOCATION.
   (let ((pin-net (arrange-pins pins (extract-internal-nets
                                       top-part)
                                     extended-top
                                     extended-right)))

       ; CALCULATES OUTER CHIP COORDINATES FOR ALL FOUR SIDES.
       ; RESULTS IN A FOUR NUMBER LIST.
     (let ((dimensions (pins-dimensions (cadr pin-net) pins
                                        pins-power ring-width
                                        extended-right
                                        extended-top)))
      (let ((top (dimensions-top dimensions))
            (right (dimensions-right dimensions))
            (left (dimensions-left dimensions))
            (bottom (dimensions-bottom dimensions)))

       ; PRODUCES RING OF PADS.
       (let ((pins-layout (place-pins (cadr pin-net) dimensions
                                      pins-power)))
        (let ((power-point (find pins-layout '(power)))
              (ground-point (find pins-layout '(ground))))
      (merge (cond ((member? 'logo option-list)
                    (move (first-quadrant
                            (title logo NM 'nonie.r.10))
                            (+ left pins-power 3)
                            (+ bottom pins-power 3)))
                   (t (null-item)))
             pins-layout

       ; PRODUCES POWER SUPPLY RAIL FOR PADS.
          (layout-power-ring pins-power power dimensions
                             extended-right extended-top
                             power-point ground-point)

       ; PRODUCES GROUND SUPPLY RAIL FOR PADS.
          (layout-ground-ring pins-power power dimensions
```

```
                        extended-right extended-top
                  power-point ground-point
                              (car pin-net))))))))))))

        ; IF PADS DO NOT FIT AROUND CURRENT CIRCUIT DIMENSIONS,
        ; AND INTENDED-RIGHT > EXTENDED-TOP, INCREASE EXTEND-TOP
        ; UNTIL ALL PADS FIT.
(def extend-right
 (lambda (pins intended-right intended-top)
  (cond
   ((< (length pins) (* 2 (+ (fix (/ intended-top
                                     (pad-class-width)))
                             (fix (/ intended-right
                                     (pad-class-width))))))
    intended-right)
   (t
    (cond
     ((< intended-right intended-top) intended-right)
     (t (* (fix (/ (1+ (- (length pins)
                          (* 2 (fix (/ intended-top
                                       (pad-class-width))))))
                   2))
           (pad-class-width)))))))))

      ; IF PADS DO NOT FIT AROUND CURRENT CIRCUIT DIMENSIONS, AND
      ; INTENDED-TOP > EXTENDED-RIGHT, INCREASE EXTENDED-TOP
      ; UNTIL ALL PADS FIT.
(def extend-top
 (lambda (pins intended-top extended-right)
  (cond
   ((< (length pins) (* 2 (+ (fix (/ intended-top
                                     (pad-class-width)))
                             (fix (/ extended-right
                                     (pad-class-width))))))
    intended-top)
   (t
    (cond
     ((<= intended-top extended-right) intended-top)
     (t (* (fix (/ (1+ (- (length pins)
                          (* 2 (fix (/ extended-right
                                       (pad-class-width))))))
                   2))
           (pad-class-width)))))))))


; EXTRACTION OF DATA FOR PAD PLACEMENT

      ; RESULTS IN A LIST CONSISTING OF TWO LISTS.  THE FIRST
      ; LIST INCLUDES ALL NET POINTS THAT ARE ON THE LEFT SIDE OF
      ; THE INTERNAL CIRCUIT AND CONNECT TO PADS.  THE SECOND
      ; LIST CONTAINS ALL SUCH POINTS ON THE RIGHT SIDE.  POINTS
      ; ON BOTH LISTS ARE ORDERED BY THEIR Y-COORDINATES.
(def extract-internal-nets
 (lambda (top-part)
```

```
  (append (list (append (excise-port-drive
                         (extract-names wing-layout)())
                         (sort-by-y (get-nets top-part '(left))
                                    ()))))
          (list (sort-by-y (get-nets top-part '(right)) ())))))

      ; TAKES A LIST OF POINTS AND RETURNS THE SAME LIST SORTED
      ; BY THEIR Y-COORDINATES
(def sort-by-y
 (lambda (list sorted-list)
  (cond
   ((null list) sorted-list)
   (t (sort-by-y1 (car list) (cdr list) list sorted-list)))))

(def sort-by-y1
 (lambda (thing l list sorted-list)
  (cond
   ((null l) (sort-by-y (excise thing list)
                        (append sorted-list (caar thing))))
   ((> (point-y (car thing)) (point-y (caar l)))
    (sort-by-y1 (car l) (cdr l) list sorted-list))
   (t (sort-by-y1 thing (cdr l) list sorted-list)))))

      ; REMOVES POINTS WITH THE NAME PORT-DRIVE FROM A LIST OF
      ; POINTS.  PORT-DRIVE IS THE NAME GIVEN TO THE SIGNAL THAT
      ; CONTROLS TRI-STATE PADS.
(def excise-port-drive
 (lambda (list new-list)
  (cond
   ((null list) new-list)
   ((equal (caaar list) 'port-drive)
    (excise-port-drive (cdr list) new-list))
   (t (excise-port-drive (cdr list) (append new-list
                                            (car list)))))))

      ; MAKES A LIST OF THE POINT NAMES OF ALL POINTS IN A LIST
      ; WITH THE ATTRIBUTE 'RING'.  THIS ATTRIBUTE IDENTIFIES
      ; POINTS INVOLVED IN PAD ROUTING.
(def extract-names
 (lambda (item)
  (append
   (setify (alpha (function point-name)
                  (find-attributes item '(ring)))))))

      ; GIVEN A LIST OF NET NAMES, IT EXTRACTS EVERY OCCURRENCE
      ; OF THOSE POINTS FROM A LIST OF POINTS.
(def get-nets
 (lambda (list side)
  (get-nets1 list (get-names list side) ())))

(def get-nets1
 (lambda (list net-names output)
  (cond
   ((null net-names) output)
```

```
   (t (get-nets1 list
         (cdr net-names)
         (append output
                 (list (find-all list
                                 (car net-names)))))))))

      ; EXTRACTS NAMES OF POINTS HAVING A SPECIFIED ATTRIBUTE.
      ; IN THIS INSTANCE THE ATTRIBUTE COULD BE 'LEFT' OR
      ;'RIGHT'.
(def get-names
 (lambda (item side)
  (setify (alpha (function point-name)
                 (find-attributes (get-ring-net item) side)))))

      ; EXTRACTS EVERY POINT WITH THE ATTRIBUTE 'RING' FROM
      ; A LIST OF POINTS.
(def get-ring-net
 (lambda (item)
  (get-ring-net1 item
          (extract-names item)
             ())))

(def get-ring-net1
 (lambda (item name-list internal-connections)
  (cond
   ((null name-list) (append
                '(nil nil nil nil)
                      (list internal-connections)))
   (t
    (get-ring-net1 item
             (cdr name-list)
             (append internal-connections
                      (find-all item (car name-list))))))))


; PIN PLACEMENT

      ; USING THE INFORMATION PROVIDED BY EXTRACT-INTERNAL-NETS,
      ; ARRANGE-PINS CONSTRUCTS A LIST THAT SPECIFIES PAD
      ; LOCATIONS.  IT FIRST TRIES TO MINIMIZE CHIP AREA BY
      ; PLACING PADS IN THE LEAST NUMBER OF SIDES.  THEN, IT
      ; COMBINES THE LIST OF POINTS FROM EXTRACT-INTERNAL-NETS
      ; WITH THE CLOCK, POWER AND GROUND-PADS.
(def arrange-pins
 (lambda (pins sorted-pins extended-top extended-right)
  (let ((left (car sorted-pins))
        (right (cadr sorted-pins)))
   (merge-common-side-lists
    (cond
     ((>= (* extended-right 2)
          (* (pad-class-width) (+ 5 (length left)
                                  (length right))))
      (append (list 2)
              (list (order-pins pins
```

```
                        (make-top-and-bottom-pin-lists
                          (reverse left)
                          right
                          '((phic) (phib) (phia))
                          (- (length left) (length right)))))))
            (t
          (cond
           ((>= (+ extended-top extended-top extended-right)
                (* (pad-class-width) (+ 5 (length left)
                                        (length right))))
            (append (list 3)
                    (list (order-pins pins
                                      (append (pre-number-pins
                                               (order-left left nil)
                                               () 'left)
                                              (pre-number-pins
                                               (order-right right
                                                            nil)
                                               () 'right))))))
           ((>= (* 2 (+ extended-top extended-right))
                (* (pad-class-width) (length pins)))
            (append (list 4)
                    (list (order-pins pins
                                      (append
                                       (pre-number-pins
                                        (order-left
                                         (append left
                                                 '((phia) (phib)
                                                   (phic)))
                                         'set)
                                        () 'left)
                                       (pre-number-pins
                                        (order-right
                                         (append '((power)) right
                                                 '((ground)))
                                         'set)
                                        () 'right)))))))))))


        ; MERGES LISTS PERTAINING TO THE SAME SIDE.  FOR EXAMPLE:
        ; (((...1) (...2) TOP) ((...3) (...4) TOP)) WOULD RESULT IN
        ; ((...1) (...2) (...3) (...4) TOP).  THE FUNCTION FINDS
        ; WHICH LISTS TO MERGE BASED ON THE NUMBER OF SIDES SLATED
        ; FOR PAD PLACEMENT.  WITH THIS INFORMATION, THE FUNCTION
        ; KNOWS WHERE IN THE NET LIST THE INDIVIDUAL LIST SEGMENTS
        ; ARE LOCATED. (LENGTH (CADR NETS)) RETURNS THE NUMBER OF
        ; PAD LISTS IN THE LIST.  SINCE THE LIST IS OF FORM:
        ; (NIL NIL NIL NIL NIL NIL NIL (.. TOP) (.. RIGHT)... ),
        ; A VALUE OF 7 INDICATES NO PIN-LISTS.
(def merge-common-side-lists
 (lambda (nets)
  (cond
   ((= (car nets) 3) ;(CAR NETS) = NUMBER OF SIDES PINS ARE
                     LOCATED
```

```
    (cond
     ((= (length (cadr nets)) 11)       ;7=NIL 4 ACTUAL PIN-LISTS
           (merge-side-lists nets 'top))
     (t nets)))
   (t
    (cond
     ((= (length (cadr nets)) 13)       ;7=NIL 6 ACTUAL PIN_LISTS
           (merge-top-and-bottom-lists nets))
     ((= (length (cadr nets)) 12)       ;7=NIL 5 ACTUAL PIN_LISTS
           (merge-top-or-bottom-lists nets))
     (t nets))))))

      ; MERGES LISTS IN A LIST WITH THE SAME 'SIDE' ATTRIBUTE.
(def merge-side-lists
 (lambda (item side)
  (append (list (car item))
          (list (append (find-net (cadr item) side)
                (delete-net-lists (cadr item) side ()))))))

      ; USED WHEN PADS ARE PLACED ON THE TOP AND BOTTOM ONLY.
(def merge-top-and-bottom-lists
 (lambda (item)
  (append (list (car item))
       (list (append (find-net (cadr item) 'top)
                (find-net (cadr item) 'bottom)
                (delete-net-lists
                  (delete-net-lists (cadr item) 'top ())
                  'bottom ()))))))

      ; WHEN ONLY THREE SIDES ARE USED TO PLACE PADS, THIS
      ; FUNCTION LOOKS FOR TWO INSTANCES OF TOP OR BOTTOM
      ; OCCURRING.
(def merge-top-or-bottom-lists
 (lambda (item)
  (cond
   ((null (cddr (find-net (cadr item) 'top)))
   (merge-side-lists item 'top))
   (t (merge-side-lists item 'bottom)))))


      ; GIVEN A LIST AND A PARAMETER, IT RETURNS 'T' IF THE
      ; PARAMETER EXISTS IN THE LIST, NIL OTHERWISE.  USED
      ; BY MERGE-TOP-OR-BOTTOM-LISTS TO DETERMINE IF THE
      ; COMMON SIDE PARAMETER IS TOP OR BOTTOM
(def find-net
 (lambda (item side)
  (find-net1
    (such-that item
           (flambda (element)
           (equal side (car (reverse element))))))))

(def find-net1
 (lambda (item)
  (list (append (cdr (reverse (car item)))
```

```
            (cadr item)))))

      ; DELETES THE SECOND LIST APPEARING WITH THE 'SIDE'
      ; ATTRIBUTE.
(def delete-net-lists
 (lambda (item side out)
  (cond
   ((null item) out)
   (t
    (cond
     ((equal side (car (reverse (car item))))
      (delete-net-lists (cdr item) side out))
     (t
      (delete-net-lists (cdr item)
                        side
                        (append out
                        (list (car item)))))))))))


      ; USED WHEN ALL PINS CAN FIT ON TOP AND BOTTOM SIDES
      ; ONLY.  CLOCK PADS ARE DISTRIBUTED AMONG THE LEFT AND
      ; RIGHT LISTS IN AN EFFORT TO EQUALIZE THE NUMBER OF
      ; PADS IN THOSE LISTS.  THE PARAMETER TIMES, THE
      ; DIFFERENCE IN THE NUMBER OF ELEMENTS BETWEEN THE
      ; ORIGINAL LEFT AND RIGHT LISTS, DETERMINES THE ACTUAL
      ; DISTRIBUTION.  POWER IS ALWAYS APPENDED TO THE LEFT LIST,
      ; AND GROUND IS ALWAYS APPENDED TO THE RIGHT LIST.  THE
      ; LEFT LIST IS PLACED ALONG THE BOTTOM.  IF THE PADS DON'T
      ; FIT, THE EXCESS IS PLACED ON THE LEFT CORNER OF THE TOP
      ; SIDE. PADS THAT CONNECT TO THE RIGHT SIDE ARE PLACED
      ; ALONG THE TOP.  AGAIN, IF THEY DON'T FIT, THE EXCESS IS
      ; PLACED ALONG THE RIGHT CORNER OF THE BOTTOM SIDE.
(def make-top-and-bottom-pin-lists
 (lambda (left right phi-list times)
  (cond
   ((= times 0)
    (cond
     ((null phi-list)
      (append (list (number-pins (append left '((power)))
                                 'bottom 'left 1))
              (list (number-pins (append right '((ground)))
                                 'top 'right
                                 (fix (/ extended-right
                                         (pad-class-width)))))))
     (t
      (cond
       ((= (length phi-list) 2)
        (append (list (number-pins (append left (cadr phi-list)
                                           '((power)))
                                   'bottom 'left 1))
                (list (number-pins (append right '((ground))
                                           (car phi-list))
                                   'top 'right
                                   (fix
```

```
                                    (/ extended-right
                                       (pad-class-width)))))))
     (t
      (append (list (number-pins (append phi-list left
                                         '((power)))
                                 'bottom 'left 1))
              (list (number-pins
                     (append right '((ground)))
                     'top 'right
                     (fix (/ extended-right
                             (pad-class-width))))))))))
((> times 0)
 (cond
  ((null phi-list)
   (cond
    ((and
      (< (* (length left) (pad-class-width)) extended-right)
      (< (* (length right) (pad-class-width)) extended-right))
     (append (list (number-pins (append left '((power)))
                                'bottom 'left 1))
             (list (number-pins
                    (append right '((ground)))
                    'top 'right
                    (fix (/ extended-right
                            (pad-class-width)))))))
    (t
     (make-top-and-bottom-pin-lists (reverse
                                     (cdr (reverse left)))
                                    (append
                                     right
                                     (list
                                      (car (reverse left))))
                                     ()
                                     (if (= times 1)
                                         0
                                         (- times 2))))))
  (t
   (append (list (number-pins (append left '((power)))
                              'bottom 'left 1))
           (list (number-pins
                  (append phi-list '((ground)) right)
                  'top 'right
                  (fix (/ extended-right
                          (pad-class-width))))))))))
(t
(cond
 ((null phi-list)
  (cond
   ((and (< (* (length left) (pad-class-width))
            extended-right)
         (< (* (length right) (pad-class-width))
            extended-right))
    (append (list (number-pins (append left '((power)))
                               'bottom 'left 1))
```

```
                (list (number-pins (append right '((ground)))
                                   'top 'right
                                    (fix
                                     (/ extended-right
                                        (pad-class-width))))))))
       (t
        (make-top-and-bottom-pin-lists (append (list (car right))
                                               left)
                                       (cdr right)
                                       ()
                                       (if (= times -1)
                                           0
                                           (+ 2
                                              times))))))
      (t
       (make-top-and-bottom-pin-lists (append left (car phi-list))
                                      right
                                      (cdr phi-list)
                                      (+ 1 times)))))))

       ; WHEN PAD PLACEMENT IS TO OCCUR ON THREE OR FOUR SIDES,
       ; IF THE PAD LIST FITS ON THE LEFT SIDE, ORDER-SIDE IS
       ; CALLED TO PLACE ALL PADS ON THE LEFT SIDE.  OTHERWISE,
       ; PADS ARE PLACED ALONG THE BOTTOM UNTIL A COUNTER REACHES
       ; A VALUE EQUAL TO HALF THE EXCESS OF PADS.  NEXT
       ; ORDER-SIDE IS CALLED TO PLACE PADS UNTIL THE LEFT SIDE IS
       ; FILLED. ANY REMAINING PADS ARE PLACED ALONG THE TOP.
(def order-left
 (lambda (list flag)
  (let ((topsize (fix (/ extended-top (pad-class-width)))))
    (cond
    ((eq flag 'set)
     (order-bottom list
                   (fix (/ (- (length list) topsize) 2))
                   0 () 'left))
    (t
     (order-side list 1 () () 'left))))))

       ; SEE COMMENTS FOR ORDER-LEFT
(def order-right
 (lambda (list flag)
  (let ((topsize (fix (/ extended-top (pad-class-width)))))
    (cond
    ((eq flag 'set)
     (order-bottom list
                   (fix (/ (- (length list) topsize) 2))
                   0 () 'right))
    (t
     (order-side list 1 () () 'right))))))


       ; PLACES PADS ON EITHER THE RIGHT OR LEFT SIDES UNTIL
       ; THE COUNTER INSTANCE ADVANCES TO TOP-SIZE.  TOP-SIZE
       ; EQUALS THE NUMBER OF PADS THAT FIT ON THE LEFT/RIGHT
```

```
      ; SIDE.
(def order-side
 (lambda (side-list instance out out1 flag)
  (cond
   ((= instance topsize)
    (cond
     ((null side-list)
      (cond
       ((eq flag 'left)
        (order-top ()
                   (append (list (append (list out1) '(left)))
                           out)
                   ()))
       (t (append (list (cons '(right) out))))))
     (t
      (cond
       ((eq flag 'left)
        (order-top (cdr side-list)
             (append (list
                      (append (list (append out1
                                            (list
                                             (car side-list))))
                              '(left)))
                      out)
             ()))
       (t
        (order-top (cdr side-list)
                   (append (list
                            (cons (append out1
                                          (list (car side-list)))
                                  '(right)))
                           out) ()))))))))
   (t
    (reverse (order-side (cdr side-list)
                         (1+ instance)
                         out
                         (append out1 (list (car side-list)))
                      flag)))))

      ; PLACES PADS ALONG THE BOTTOM SIDE UNTIL THE COUNTER
      ; INSTANCE = TIMES.
(def order-bottom
 (lambda (side-list times instance out flag)
  (cond
   ((= instance times)
    (order-side side-list
                1
                (list (append (list out) '(bottom)))
                ()
                flag))
   (t
    (order-bottom (cdr side-list)
                  times
                  (1+ instance)
```

```
                (append out (list (car side-list)))
                flag)))))


      ; PLACES PADS ALONG THE TOP UNTIL THE PAD LIST IS
      ; EXHAUSTED.
(def order-top
 (lambda (side-list out out1)
  (cond
   ((null side-list)
    (cond
     ((eq 'left (car (reverse (car out))))
      (append (list (append (list out1)
                            '(top)))
              out))
     (t
      (append (list (cons out1 '(top))) out))))
   (t
    (cond
     ((eq 'left (car (reverse (car out))))
      (order-top (cdr side-list)
                 out
                 (append out1 (list (car side-list)))))
     (t
      (order-top (cdr side-list)
                 out
                 (append (list (car side-list)) out1)))))))))


      ; PINS IS A LIST CONTAINING FULL PAD NAMES.  SORTED-PINS
      ; IS A LIST CONTAINING POINTS.  ORDER-PINS RETURNS
      ; A LIST OF PAD NAMES IN THE ORDER OF SORTED-LIST.
      ; THIS IS NECESSARY BECAUSE THE FUNCTION PAD-LAYOUT
      ; REQUIRES PAD NAMES TO FULLY IDENTIFY A PAD.
(def order-pins
 (lambda (pins sorted-pins)
  (cond
   ((null sorted-pins) (null-item))
   (t
    (append
     (list (excise (null-item)
                   (order-pins1 pins
                                (cdr (reverse (car sorted-pins)))
                                  (car (reverse
                                        (car sorted-pins))))))
     (order-pins pins (cdr sorted-pins)))))))

(def order-pins1
 (lambda (pins work-list side)
  (cond
   ((null work-list) (list side))
   (t
    (append (list (order-pins2 pins (car work-list) side))
            (order-pins1 pins (cdr work-list) side))))))
```

```
(def order-pins2
 (lambda (pins work-list side)
  (cond
   ((or (is-pad-output4? (cadar pins))
        (is-pad-output8? (cadar pins)))
    (cond
     ((and (eq (cadaddadr (car pins)) (cadar work-list))
           (eq (caddaddadr (car pins)) (caddar work-list)))
      (append (cdr work-list) (cdar pins)))
     (t
      (order-pins2 (cdr pins) work-list side))))
   ((or (is-pad-tri-state4? (cadar pins))
        (is-pad-tri-state8? (cadar pins)))
    (cond
     ((and (eq (cadaddadr (car pins)) (cadar work-list))
           (eq (caddaddadr (car pins)) (caddar work-list)))
      (append (cdr work-list) (cdar pins)))
     (t
      (order-pins2 (cdr pins) work-list side))))
   ((or (is-pad-power? (cadar pins))
        (is-pad-ground? (cadar pins))
        (is-pad-phia? (cadar pins))
        (is-pad-phib? (cadar pins))
        (is-pad-phic? (cadar pins)))
    (cond
     ((eq (caadr (car pins)) (caar work-list))
      (append (cdr work-list) (cdar pins)))
     (t
      (order-pins2 (cdr pins) work-list side))))
   ((is-pad-input? (cadar pins))
    (cond
     ((if (= (length (car (reverse (cadr (car pins))))) 2)
            (eq (cadar (reverse (cadr (car pins))))
                (cadar work-list))
            (and (eq (cadar (reverse (cadr (car pins))))
                     (cadar work-list))
                 (eq (caddar (reverse (cadr (car pins))))
                     (caddar work-list))))
      (append (cdr work-list) (cdar pins)))
     (t
      (order-pins2 (cdr pins) work-list side))))
   (t (null-item)))))


      ; PRE-NUMBER-PINS AND PRE-NUMBER-PINS1 ARE USED WHEN
      ; PADS ARE PLACED ON THREE OR FOUR SIDES.  THESE
      ; FUNCTION ENSURE THAT THE INTERNAL TRMINATION SITE,
      ; LEFT OR RIGHT, ARE CONSIDERED IN NUMBER ASSIGNMENT.
      ; FOR EXAMPLE, PADS ALONG THE TOP THAT CONNECT TO THE
      ; RIGHT SIDE OF THE CIRCUIT SHOULD BE POSITIONED ON
      ; THE RIGHT CORNER, WHILE THOSE THAT CONNECT TO THE LEFT
      ; SIDE ARE PLACED ON THE LEFT CORNER.
(def pre-number-pins
```

```
 (lambda (list out flag)
  (cond
   ((null list) out)
   (t
    (append out (pre-number-pins1 (cdr list) (caar list)
                                  (cadar list) flag))))))

(def pre-number-pins1
 (lambda (list side-list side1 flag)
  (cond
   ((and (or (eq side1 'top) (eq side1 'bottom))
         (eq flag 'right))
    (pre-number-pins list
                     (list (number-pins
                            (if (eq flag 'top)
                                side-list
                                (reverse side-list))
                            side1
                            flag
                            (fix (/ extended-right
                                    (pad-class-width)))))
                     flag))
   (t
    (pre-number-pins list
                     (list (number-pins side-list side1
                                        flag 1))
                     flag)))))

      ; ASSIGNS A NUMBER TO EACH PIN THAT, ALONG WITH THE SIDE,
      ; LOCATES EACH PAD.  NUMBERS FOR PINS ON THE TOP OR
      ; BOTTOM SIDES THAT CONNECT TO THE LEFT SIDE ARE ASSIGNED
      ; BY A COUNTER STARTING AT 0.  THE COUNTER FOR PADS THAT
      ; CONNECT TO THE RIGHT SIDE STARTS WITH THE MAXIMUM NUMBER
      ; (THE PAD AT THE RIGHT CORNER) AND COUNTS DOWN UNTIL THE
      ; LIST IS EXHAUSTED.
(def number-pins
 (lambda (list side1 side2 pin-number)
  (cond
   ((null list) (list side1))
   ((and (or (eq side1 'top) (eq side1 'bottom))
         (eq side2 'right))
    (append (list (cons (car list) (list pin-number)))
            (number-pins (cdr list) side1 side2
                         (1- pin-number))))
   (t
    (append (list (cons (car list) (list pin-number)))
         (number-pins (cdr list) side1 side2
                      (1+ pin-number)))))))

      ; PINS-DIMENSIONS DETERMINES THE DIMENSIONS OF THE CIRCUIT.
      ; IT'S OUTPUT IS A FOUR NUMBER VECTOR CONTAINING THE
      ; POSITIONS OF THE TOP, RIGHT BOTTOM AND LEFT SIDES OF THE
      ; CIRCUIT.  VARIOUS PARAMETERS ARE CONSIDERED IN
      ; CALCULATING THESE NUMBERS.  AMONG THEM:
```

```
      ;     1.  MAXIMUM PAD-HEIGHT FOUND IN THAT SIDE
      ;     2.  SIZE OF THE INTERNAL CIRUIT LAYOUT
      ;     3.  POWER AND GROUND RING REQUIREMENTS
      ;     4.  RING-WIDTH
(def pins-dimensions
 (lambda (pin-net pins power ring-width extended-right
                  extended-top)
  (cond

         ; WHEN PADS PLACED ON TOP AND BOTTOM SIDES ONLY
   ((>= (* 2 extended-right) (* (length pins) (pad-class-width)))
    (make-dimensions
     0
     (+ extended-top (ring-width-top ring-width) power power 3

                ; RETURNS THE HEIGHT OF THE TALLEST PAD ON
                ; THE TOP PAD LIST
              (slash-alpha (cdr (reverse (cadr pin-net)))
                       (pad-class-basic-height)
                           (function max)
                           (flambda (pin) (pin-height pin
                                                      power))))
     (+ extended-right (ring-width-right ring-width) power 3)
     (- 0 (ring-width-bottom ring-width) pins-power power power 3
                ; RETURNS THE HEIGHT OF THE TALLEST PAD ON
                ; THE BOTTOM PAD LIST
          (slash-alpha (cdr (reverse (car pin-net)))
                       (pad-class-basic-height)
                       (function max)
                       (flambda (pin) (pin-height pin power))))
     (- 0 (ring-width-left ring-width) power power 6)))
   ((>= (+ extended-top extended-top extended-right)
        (* (length pins) (pad-class-width)))
    (make-dimensions
     0
     (+ extended-top (ring-width-top ring-width) power power 3
        (slash-alpha (cdr (reverse (car pin-net)))
                     (pad-class-basic-height)
                     (function max)
                     (flambda (pin) (pin-height pin power))))
     (+ extended-right (ring-width-right ring-width)
        power power 3 (slash-alpha
                       (cdr (reverse (caddr pin-net)))
                       (pad-class-basic-height)
                       (function max)
                       (flambda (pin) (pin-height pin power))))
     (- 0 (ring-width-bottom ring-width) power power 3)
     (- 0 (ring-width-left ring-width) power power 3
          (slash-alpha (cdr (reverse (cadr pin-net)))
                       (pad-class-basic-height)
                       (function max)
                       (flambda (pin) (pin-height pin power))))))
   (t
    (make-dimensions
```

```
     0
     (+ extended-top (ring-width-top ring-width) power power 3
     (slash-alpha (cdr (reverse (car pin-net)))
                  (pad-class-basic-height)
                  (function max)
                  (flambda (pin) (pin-height pin power))))
     (+ extended-right (ring-width-right ring-width)
        power power 3 (slash-alpha
                        (cdr (reverse (cadddr pin-net)))
                        (pad-class-basic-height)
                        (function max)
                        (flambda (pin) (pin-height pin power))))
     (- 0 (ring-width-bottom ring-width) power power 3
          (slash-alpha (cdr (reverse (cadr pin-net)))
                        (pad-class-basic-height)
                        (function max)
                        (flambda (pin) (pin-height pin power))))
     (- 0 (ring-width-left ring-width) power power 3
          (slash-alpha (cdr (reverse (caddr pin-net)))
                        (pad-class-basic-height)
                        (function max)
                        (flambda (pin) (pin-height
                                        pin power)))))))))

      ; GIVEN THE COMPLETE PAD NETS, PLACE-PINS BREAKS OFF THE
      ; LIST OF PADS FOR ONE SIDE AND GIVES IT TO PLACE-PINS1
      ; FOR FURTHER PROCESSING.  DIMENSIONS VALUES ARE OBTAINED
      ; FROM PINS-DIMENSIONS.  POWER IS THE WIDTH OF THE SKELETON
      ; POWER/GROUND RAILS.
(def place-pins
 (lambda (pin-list dimensions power)
  (cond
   ((= (length pin-list) 7) (null-item))
   (t
    (merge (place-pins1 (cdr (reverse (car pin-list)))
                        (car (reverse (car pin-list)))
                        dimensions power)
           (place-pins (cdr pin-list) dimensions power))))))

      ; GIVEN A LIST OF PADS ON A GIVEN SIDE, PLACE-PINS1 PEELS
      ; OFF INDIVIDUAL PADS AND GICES THEM TO PLACE-PIN FOR
      ; FURTHER PROCESSING.
(def place-pins1
 (lambda (pin-list1 side dimensions power)
  (let ((top (dimensions-top dimensions))
        (right (dimensions-right dimensions))
        (bottom (dimensions-bottom dimensions))
        (left (dimensions-left dimensions)))
   (cond
    ((null pin-list1) (null-item))
    (t
     (merge (place-pin (cadar pin-list1)
                       (caar pin-list1) side power)
            (place-pins1 (cdr pin-list1) side
```

```
                    dimensions power)))))))

      ; PLACES AND ORIENTS THE PIN LAYOUTS.  THE ACTUAL
      ; LAYOUT IS PRODUCED BY LAYOUT-PAD.  ORIGINAL PAD
      ; ORIENTATION IS SUITED FOR LAYING PADS ON THE BOTTOM.
      ; TO OBTAIN CORRECT ORIENTATIONS FOR THE OTHER SIDES,
      ; THE L5 FUNCTION MIRRORX (PRODUCES AN IDENTICAL IMAGE
      ; AS IF THE X-AXIS WERE A MIRROR) IS USED FOR THE TOP,
      ; THE L5 FUNCTION ROTCCW (ROTATE COUNTER CLOCKWISE) IS
      ; USED FOR THE RIGHT SIDE,
      ; THE L5 FUNCTION ROTCW (ROTATE CLOCKWISE) IS USED FOR
      ; THE LEFT SIDE.
      ; THE PRODUCT OF THE PIN-NUMBER AND THE WIDTH OF THE
      ; PAD-CLASS PROVIDES THE X-COORDINATE POSITION WHEN
      ; PLACING PADS ON THE TOP OR BOTTOM SIDES, AND THE
      ; Y-COORDINATE FOR THE RIGHT AND LEFT SIDES.
(def place-pin
 (lambda (pin pin-number side power)
   (cond
    ((eq side 'top)
     (move (mirrorx (layout-pad pin power 'top))
           (* (pad-class-width) (1- pin-number))
           top))
    ((eq side 'right)
     (move (rotccw (layout-pad pin power 'right))
           right
           (* (pad-class-width) (1- pin-number))))
    ((eq side 'bottom)
     (move (layout-pad pin power 'bottom)
           (* (pad-class-width) (1- pin-number))
           bottom))
    (t
     (move (rotccw (mirrorx (layout-pad pin power 'left)))
           left
           (* (pad-class-width) (1- pin-number)))))))

      ; LAYOUT-GROUND-RING BUILDS A METAL1 RING ON THE INTERIOR
      ; PAD BOUNDARY AND CONNECTS IT TO THE SKELETON GROUND RAIL
      ; SITUATED ON THE TOP OF THE INTERNAL CIRCUIT LAYOUT.  THE
      ; THE LAYOUT DEPENDS ON THE NUMBER OF SIDES USED TO PLACE
      ; PADS, THE SIZE OF THE INTERNAL LAYOUT, THE DIMENSIONS
      ; OBTAINED FROM PINS-DIMENSIONS, AND THE LOCATION OF THE
      ; POWER PAD. THIS ROUTINE WILL ONLY LAY METAL WHERE
      ; REQUIRED.  FOR LAYOUTS WITH PADS ON 2 OR 3 SIDES, THE
      ; RING CONSISTS OF WIRES ON THE LEFT, RIGHT AND TOP SIDES
      ; WITH A CONNECTION TO GROUND-PAD.  FOR LAYOUTS WITH PADS
      ; ON 4 SIDES, THE RING CONSISTS OF WIRES ON ALL FOUR SIDES.
(def layout-ground-ring
 (lambda (pins-power power dimensions extended-right extended-top
          power-point ground-point sides)
  (let ((offset (+ pins-power (pad-class-basic-height))))
   (let ((top (- (dimensions-top dimensions) offset))
         (right (- (dimensions-right dimensions) offset))
         (bottom (+ (dimensions-bottom dimensions) offset))
```

```
(left (+ (dimensions-left dimensions) offset)))
 (cond
  ((= sides 2)
   (merge
     (rect 'NM (- (point-x ground-point) (/up power 2))
        intended-top
               (+ (point-x ground-point) (/up power 2))
               (point-y ground-point))
     (rect 'NM (+ (- left offset) pins-power 3)
               (- top pins-power)
               extended-right
               top)
     (rect 'NM (+ (- left offset) pins-power 3)
               bottom
               (- (point-x power-point) (/up (pad-class-width)
                                             2))
               (+ bottom pins-power))
     (rect 'NM (+ (- left offset) pins-power 3)
               bottom
               (+ (- left offset) pins-power pins-power 3)
               top)))
  ((= sides 4)
   (merge
    (rect 'NM (- (point-x ground-point) (/up power 2))
              intended-top
              (+ (point-x ground-point) (/up power 2))
              (point-y ground-point))
    (rect 'NM left
              bottom
              (+ left pins-power)
              top)
    (rect 'NM left
              (- top pins-power)
              right
              top)
    (rect 'NM (- right pins-power)
              bottom
              right
              top)
    (rect 'NM left
              bottom
              (- (point-x power-point) (/up (pad-class-width)
                                            2))
              (+ bottom pins-power))
    (rect 'NM (+ (point-x power-point) (/up (pad-class-width)
                                            2))
              bottom
              right
              (+ bottom pins-power))))
  (t
   (merge
    (rect 'NM (- (point-x ground-point) (/up power 2))
              intended-top
              (+ (point-x ground-point) (/up power 2))
```

```
                (point-y ground-point))
        (rect 'NM left
                bottom
                (+ left pins-power)
                top)
        (rect 'NM left
                (- top pins-power)
                right
                top)
        (rect 'NM (- right pins-power)
                (+ bottom (pad-class-width))
                right
                top)))))))


      ; RETURNS A LIST OF FOUR NUMBERS THAT INDICATE THE WIDTH
      ; REQUIRED BY THE TOP, RIGHT, BOTTOM AND LEFT PAD ROUTING
      ; CHANNELS.
(def get-ring-width
 (lambda (item right top)
  (make-ring-width
   (* 7 (if (null (extract-nets item 'top right top))
            0
            (net-track-number
             (minmax (extract-nets item 'top right top)
                          (flambda (net1 net2)
                             (> (net-track-number net1)
                                (net-track-number net2)))))))
   (* 7 (if (null (extract-nets item 'right right top))
            0
            (net-track-number
             (minmax (extract-nets item 'right right top)
                           (flambda (net1 net2)
                             (> (net-track-number net1)
                                (net-track-number net2)))))))
   (* 7 (if (null (extract-nets item 'bottom right top))
            0
            (net-track-number
             (minmax (extract-nets item 'bottom right top)
              (flambda (net1 net2)
                           (> (net-track-number net1)
                              (net-track-number net2)))))))
   (* 7 (if (null (extract-nets item 'left right top))
            0
            (net-track-number
             (minmax (extract-nets item 'left right top)
                     (flambda (net1 net2)
                           (> (net-track-number net1)
                              (net-track-number net2)))))))))))


; NET EXTRACTION

      ; GIVEN A LIST OF POINTS RETURNS A LIST OF THE POINT
```

```
      ; Y-COORDINATES, ORDERED FROM LEAST TO GREATEST.
(def sort-y
 (lambda (list sorted-list)
  (cond
   ((null list) sorted-list)
   (t (sort-y1 (car list) (cdr list) list sorted-list)))))

(def sort-y1
 (lambda (thing l list sorted-list)
  (cond
   ((null l) (sort-y (excise thing list)
    (append sorted-list (list (point-y (car thing))))))
   ((> (point-y (car thing)) (point-y (caar l)))
    (sort-y1 (car l) (cdr l) list sorted-list))
   (t (sort-y1 thing (cdr l) list sorted-list)))))

      ; PREP-PAD-BANK IDENTIFIES THE LOCATION OF THE GROUND
      ; AND POWER PADS AND CALLS ON LEFT-PAD-BANK AND
      ; RIGHT-PAD-BANK TO BUILD THE PAD NET LIST.
(def prep-pad-bank
 (lambda (list side)
  (let ((ground-point (point-x (find pins-layout '(ground))))
    (power-point (point-x (find pins-layout '(power)))))
   (cond
    ((eq side 'left)
     (left-pad-bank (get-nets list '(bottom))
                    (get-nets list '(left))
                    (get-nets list '(top))))
    (t
     (right-pad-bank (get-nets list '(bottom))
                     (get-nets list '(right))
                     (get-nets list '(top)))))))))


      ; IDENTIFIES THOSE PADS LOCATED TO THE LEFT OF THE
      ; POWER PAD, IF ON THE BOTTOM, AND TO THE LEFT OF
      ; THE GROUND PAD, IF ON THE TOP AND RETURNS A LIST
      ; THAT PINPOINTS THEIR LOCATION AS FOLLOWS:
      ;    1.  IF PAD IS ON THE LEFT SIDE,
      ;        USE THE PAD'S Y-COORDINATE
      ;    2.  IF PAD IS ON THE BOTTOM SIDE,
      ;        USE THE PAD X-COORDINATE * -1
      ;    3.  IF PAD IS ON TOP,
      ;        USE THE PAD X-COORDINATE + EXTENDED-TOP.
(def left-pad-bank
 (lambda (bottom left top)
  (append
   (reverse
    (map-argument 'times -1
                  (extract-points bottom power-point '< ())()))
    (mapcar 'caddar left)   ;;CADDR = POINT-Y
    (map-argument 'plus extended-top
                  (extract-points top ground-point '< ())()))))
```

```
      ; IDENTIFIES THOSE PADS LOCATED TO THE RIGHT OF THE
      ; POWER PAD, IF ON THE BOTTOM, AND TO THE RIGHT OF
      ; THE GROUND PAD, IF ON THE TOP AND RETURNS A LIST
      ; THAT PINPOINTS THEIR LOCATION AS FOLLOWS:
      ;    1.  IF PAD IS ON THE RIGHT SIDE,
      ;        USE THE PAD'S Y-COORDINATE
      ;    2.  IF PAD IS ON THE BOTTOM SIDE,
      ;        USE THE PAD X-COORDINATE * -1
      ;    3.  IF PAD IS ON TOP,
      ;        USE THE PAD X-COORDINATE + EXTENDED-TOP.
(def right-pad-bank
 (lambda (bottom right top)
  (append
   (map-argument 'times -1
                 (extract-points bottom power-point '> ()) ())
   (mapcar 'caddar right)
   (map-argument 'plus extended-top
                 (extract-points top ground-point '> ()) ()))))

      ; EXTRACTS FROM A LIST OF POINTS THOSE POINTS THAT MEET
      ; THE CONDITION SET BY PREDICATE WITH RESPECT TO THE
      ; PARAMETER POINT.
(def extract-points
 (lambda (list point predicate output)
  (cond
   ((null list) output)
   (t
    (cond
     ((predicate (cadaar list) point)
      (extract-points (cdr list) point predicate
                      (append output (list (cadaar list)))))
     (t (extract-points (cdr list) point predicate output)))))))

      ; APPLIES THE FUNCTION SPECIFIED BY PREDICATE AND THE
      ; ARGUMENT SPECIFIED BY THE PARAMETER ARGUMENT TO EVERY
      ; ELEMENT OF A LIST.  fOR EXAMPLE,
      ; (MAP-ARGUMENT '+ 4 LIST), ADDS 4 TO EVERY ELEMENT IN
      ; LIST.
(def map-argument
 (lambda (predicate argument list output)
  (cond
   ((null list) output)
   (t (map-argument predicate argument (cdr list)
                    (append output
                            (list (predicate (car list)
                                             argument)))))))))


; NET LAYOUT

      ; TOP-LEVEL OF PAD ROUTING ROUTINES.  THE PROBLEM IS
      ; DIVIDED IN TWO; THE LEFT AND RIGHT ROUTING PROBLEM.
      ; MOAT BREAKS THE NET-LISTS IN TWO.  INNER-BANK CONTAINS
      ; THE INTERNAL NET TERMINALS AND OUTER-BANK THE PAD
```

```
      ; TERMINALS. THE NET-LISTS ARE PASSED TO ROUTE-LEFT-BOTTOM
      ; OR ROUTE-RIGHT-BOTTOM.  THESE ROUTE ALL NETS WITH PADS ON
      ; THE BOTTOM.  WHEN FINISHED, IT PASSES WHAT REMAINS OF THE
      ; NET-LIST TO ROUTE-LEFT-SIDE OR ROUTE-RIGHT-SIDE.  THESE
      ; ROUTE NETS WITH PADS ON THE LEFT OR RIGHT SIDES.  WHAT'S
      ; LEFT OF THE NET-LISTS IS THEN ROUTED BY ROUTE-LEFT-TOP
      ; OR ROUTE-RIGHT-TOP.

(def moat
 (lambda (left-ring-nets right-ring-nets layer width ring-width)
 (declare (special layer width ring-width))
  (let ((space 3))
   (merge
    (let ((inner-bank (car left-ring-nets))
          (outer-bank (cadr left-ring-nets)))
     (route-left-bottom (car left-ring-nets)
                        (cadr left-ring-nets) 1))
      (let ((inner-bank (car right-ring-nets))
            (outer-bank (cadr right-ring-nets)))
       (route-right-bottom (car right-ring-nets)
                           (cadr right-ring-nets)
                           1))))))

      ; MACROS USED TO EXTRACT THE DESIRED NUMBER FROM
      ; RING-WIDTH.
(defmacro top-width (item)
 (list 'car item))

(defmacro right-width (item)
 (list 'cadr item))

(defmacro bottom-width (item)
 (list 'caddr item))

(defmacro left-width (item)
 (list 'cadddr item))


      ; ROUTES NETS BETWEEN PADS ON THE BOTTOM AND CORRESPONDING
      ; TERMINAL ON THE LEFT SIDE UNTIL IT ENCOUNTERS POSITIVE
      ; VALUE IN OUTER-BANK.  IT THEN PASSES THE NET-LIST TO
      ; ROUTE-LEFT-SIDE.
(def route-left-bottom
 (lambda (t-in t-out track-number)
  (cond
   ((null t-out) (null-item))
   ((> (car t-out) 0) (route-left-side t-in t-out
                                       'up track-number))
   (t
    (merge
     (rect layer
        (abs (+ (car t-out) (/ width 2)))
        (- 0  power (bottom-width ring-width))
        (abs (- (car t-out) (/ width 2)))
```

```
      (+ (- 0 (* track-number 7)) width))
    (rect layer
      (- 0 (* track-number 7))
      (- 0 (* track-number 7))
      (abs (- (car t-out) (/ width 2)))
      (+ (- 0 (* track-number 7)) width))
    (rect layer
      (- 0 (* track-number (+ space width)))
      (- 0 (* track-number (+ space width)))
      (- width (* track-number (+ space width)))
      (+ (car t-in) (/ width 2)))
    (move (poly-cut) (- 0 (* track-number (+ space width)))
                     (+ (car t-in) (/ width 2)))
    (rect 'NP (- (/ width 2) (* track-number (+ space width)))
                  (- (car t-in) 1)
              (+ power space (/ width 2))
              (+ (car t-in) 1))
    (route-left-bottom (cdr t-in) (cdr t-out)
                       (+ 1 track-number)))))))

     ; ROUTES NETS BETWEEN PADS ON THE LEFT SIDE AND THE
     ; INTERNAL CIRCUIT UNTIL IT ENCOUNTERS AN ELEMENT IN
     ; OUTER-BANK WITH A VALUE GREATER THAN EXTENDED-TOP.
     ; IT THEN PASSES WHAT REMAINS OF THE LIST TO
     ; ROUTE-LEFT-TOP.
(def route-left-side
 (lambda (t-in t-out flag track)
  (let ((span (left-width ring-width)))
  (cond
   ((null t-out) (null-item))
   ((> (car t-out) extended-top)
    (route-left-top t-in t-out))
   (t
    (cond
     ((= (car t-in) (car t-out))
      (merge
       (rect 'NM (- 0 power span)
                 (- (car t-out) (/ width 2))
                 (- 0 space)
                 (+ (car t-out) (/ width 2)))
       (move (poly-cut) (- 0 space width)
                        (+ (car t-out) (/ width 2)))
       (rect 'NP (- 0 space (/ width 2))
                 (- (car t-out) 1)
                 (+ power space (/ width 2))
                 (+ (car t-out) 1))
       (route-left-side (cdr t-in) (cdr t-out) 'straight 1)))
 ((> (car t-in) (car t-out))
      (merge
       (route-left-moat-up (car t-in) (car t-out)
                           (if (eq flag 'up) track 1))
       (route-left-side (cdr t-in) (cdr t-out) 'up
                           (if (eq flag 'up) (add1 track) 2))))
     (t
```

```
      (merge
       (route-left-moat-down (car t-in) (car t-out)
                             (if (eq flag 'down) track 1))
(route-left-side (cdr t-in) (cdr t-out) 'down
                        (if (eq flag 'down)
                            (add1 track)
                            2)))))))))))
      ; ROUTES WHAT REMAINS OF NET-LIST BETWEEN PADS ON TOP
      ; AND TERMINALS ON THE LEFT-SIDE.
(def route-left-top
 (lambda (t-in t-out)
  (let ((span (top-width ring-width))
        (stretch 0))
   (cond
    ((null t-in) (null-item))
    (t
     (merge
      (route-left-top1 t-in t-out (length t-in))
      (route-left-top (cdr t-in) (cdr t-out))))))))

(def route-left-top1
 (lambda (t-in t-out track)
  (merge
   (rect 'NM (- (car t-out) extended-top (/ width 2))
             (- (+ extended-top (* (+ space width) track))
                width)
             (+ (- (car t-out) extended-top) (/ width 2))
             (+ extended-top span power power))
   (rect 'NM (- 0 (* (+ space width) track))
             (- (+ extended-top (* (+ space width) track))
                width)
             (+ (- (car t-out) extended-top) (/ width 2))
             (+ extended-top (* (+ space width) track)))
   (rect 'NM (- 0 (* (+ space width) track))
             (- (car t-in) (/ width 2))
             (- width (* (+ space width) track))
             (+ extended-top (* ( + space width) track)))
   (move (poly-cut) (- 0 (* track (+ space width)))
                    (+ (car t-in) (/ width 2)))
   (rect 'NP (- width (/ width 2) (* track (+ space width)))
                  (- (car t-in) 1)
             (+ power 3 (/ width 2))
             (+ (car t-in) 1)))))

      ; WHEN THE TERMINAL IN INNER-BANK > TERMINAL FROM
      ; OUTER-BANK.
(def route-left-moat-up
 (lambda (inner outer track)
    (merge
     (rect 'NM (- 0 span power power)
               (- outer (/ width 2))
               (- width (* track (+ space width)))
               (+ outer (/ width 2)))
     (rect 'NM (- 0 (* track (+ space width)))
```

```
              (- outer (/ width 2))
              (- width (* track (+ space width)))
              (+ inner (/ width 2)))
     (move (poly-cut) (- 0 (* track (+ space width)))
                      (+ inner (/ width 2)))
     (rect 'NP (- 0 (* track (+ space width)) (/ width 2))
              (- inner 1)
              (+ power 3 (/ width 2))
              (+ inner 1)))))

      ; WHEN TERMINAL IN OUTER-BANK > TERMINAL FROM INNER-BANK.
(def route-left-moat-down
 (lambda (inner outer track)
    (merge
     (rect 'NM (- 0 span power power)
              (- outer (/ width 2))
              (- (* track (+ space width)) space span)
              (+ outer (/ width 2)))
     (rect 'NM (- (* track (+ space width)) span space width)
              (- inner (/ width 2))
              (- (* track (+ space width)) span space)
              (+ outer (/ width 2)))
     (move (poly-cut) (- (* track (+ space width))
                         span space width)
                      (+ inner (/ width 2)))
     (rect 'NP (- (* track (+ space width))
                  span  space (/ width 2))
              (- inner 1)
              (+ power 3 (/ width 2))
              (+ inner 1)))))

      ; ROUTES NETS BETWEEN PADS ON THE BOTTOM AND CORRESPONDING
      ; TERMINAL ON THE RIGHT SIDE UNTIL IT ENCOUNTERS POSITIVE
      ; VALUE IN OUTER-BANK.  IT THEN PASSES THE NET-LIST TO
      ; ROUTE-RIGHT-SIDE.
(def route-right-bottom
 (lambda (t-in t-out track-number)
  (cond
   ((null t-out) (null-item))
   ((> (car t-out) 0) (route-right-side t-in t-out 'up
                                        track-number))
   (t
    (merge
     (rect layer (abs (+ (car t-out) (/ width 2)))
                 (- 0 (bottom-width ring-width) power)
                 (abs (- (car t-out) (/ width 2)))
                 (+ (- 0 (* track-number 7)) width))
     (rect layer (abs (+ (car t-out) (/ width 2)))
                 (- 0 (* track-number 7))
                 (+ extended-right (* track-number 7))
                 (+ (- 0 (* track-number 7)) width))
     (rect layer (- (+ extended-right (* track-number
                                         (+ space width)))
                    width)
```

```
                (- 0 (* track-number (+ space width)))
                (+ extended-right (* track-number
                                     (+ space width)))
                (+ (car t-in) (/ width 2)))
    (move (poly-cut) (- (+ extended-right
                           (* track-number (+ space width)))
                        width)
                     (+ (car t-in) (/ width 2)))
    (rect 'NP (- extended-right power 3 power 3)
              (- (car t-in) 1)
              (+ extended-right space (/ width 2))
              (+ (car t-in) 1))
    (route-right-bottom (cdr t-in) (cdr t-out)
                        (+ 1 track-number)))))))

     ; ROUTES NETS BETWEEN PADS ON THE RIGHT SIDE AND THE
     ; INTERNAL CIRCUIT UNTIL IT ENCOUNTERS AN ELEMENT IN
     ; OUTER-BANK WITH A VALUE GREATER THAN EXTENDED-TOP.
     ; IT THEN PASSES WHAT REMAINS OF THE LIST TO
     ; ROUTE-RIGHT-TOP.
(def route-right-side
 (lambda (t-in t-out flag track)
  (let ((span (right-width ring-width)))
  (cond
   ((null t-out) (null-item))
   ((> (car t-out) extended-top) (route-right-top t-in t-out))
 (t
    (cond
     ((= (car t-in) (car t-out))
      (merge
       (rect 'NM (+ extended-right space)
                 (- (car t-in) (/ width 2))
                 (+ extended-right span power)
                 (+ (car t-in) (/ width 2)))
       (move (poly-cut) (+ extended-right space)
                        (+ (car t-in) (/ width 2)))
       (rect 'NP (- extended-right power 3 power 3 (/ width 2))
                 (- (car t-in) 1)
                 (+ extended-right space (/ width 2))
                 (+ (car t-in) 1))
       (route-right-side (cdr t-in) (cdr t-out) 'straight 1)))
 ((> (car t-in) (car t-out))
    (merge
     (route-right-moat-up (car t-in) (car t-out)
                          (if (eq flag 'up) track 1))
     (route-right-side (cdr t-in) (cdr t-out) 'up
                       (if (eq flag 'up) (add1 track) 2))))    (t
    (merge
     (route-right-moat-down (car t-in) (car t-out)
                            (if (eq flag 'down) track 1))
     (route-right-side (cdr t-in) (cdr t-out) 'down
                       (if (eq flag 'down)
                           (add1 track)
                           2)))))))))
```

```
      ; WHEN THE TERMINAL IN INNER-BANK > TERMINAL FROM
      ; OUTER-BANK.
(def route-right-moat-up
 (lambda (inner outer track)
    (merge
     (rect 'NM (+ extended-right
                  (* track (+ space width)))
               (- outer (/ width 2))
               (+ extended-right span power power)
               (+ outer (/ width 2)))
     (rect 'NM (+ extended-right
                  (* track (+ space width)))
               (- outer (/ width 2))
               (+ extended-right width
                  (* track (+ space width)))
               (+ inner (/ width 2)))
     (move (poly-cut) (+ extended-right
                         (* track (+ space width)))
                      (+ inner (/ width 2)))
     (rect 'NP (- extended-right power 3 power 3 (/ width 2))
               (- inner 1)
               (+ extended-right (/ width 2)
                  (* track (+ space width)))
               (+ inner 1)))))
      ; WHEN THE TERMINAL IN INNER-BANK < TERMINAL FROM
      ; OUTER-BANK.
(def route-right-moat-down
 (lambda (inner outer track)
    (merge
     (rect 'NM (- (+ extended-right space span)
                  (* track (+ space width)))
               (- outer (/ width 2))
               (+ extended-right span space power power)
               (+ outer (/ width 2)))
     (rect 'NM (- (+ extended-right span  space)
                  (* track (+ space width)))
               (- inner (/ width 2))
               (- (+ extended-right span width space)
                  (* track (+ space width)))
               (+ outer (/ width 2)))
     (move (poly-cut) (- (+ extended-right span space)
                         (* track (+ space width)))
                      (+ inner (/ width 2)))
     (rect 'NP (- extended-right power 3 power 3 (/ width 2))
           (- inner 1)
               (- (+ extended-right span (/ width 2) space)
            (* track (+ space width)))
               (+ inner 1)))))

      ; ROUTES WHAT REMAINS OF NET-LIST BETWEEN PADS ON TOP
      ; AND TERMINALS ON THE LEFT-SIDE.
(def route-right-top
 (lambda (t-in t-out)
  (let ((span (top-width ring-width))
```

```
    (stretch 0))
   (cond
    ((null t-in) (null-item))
    (t
     (merge
      (route-right-top1 t-in t-out (length t-in))
      (route-right-top (cdr t-in) (cdr t-out)))))))))

(def route-right-top1
 (lambda (t-in t-out track)
  (merge
   (rect 'NM (- (car t-out) extended-top (/ width 2))
             (- (+ extended-top (* (+ space width) track))
          width)
             (+ (- (car t-out) extended-top) (/ width 2))
             (+ extended-top span power power))
   (rect 'NM (- (car t-out) extended-top (/ width 2))
             (- (+ extended-top (* (+ space width) track))
                width)
             (+ extended-right (* track (+ space width)))
             (+ extended-top (* (+ space width) track)))
   (rect 'NM (- (+ extended-right (* (+ space width) track))
                width)
             (- (car t-in) (/ width 2))
             (+ extended-right (* (+ space width) track))
             (+ extended-top (* ( + space width) track)))
   (move (poly-cut) (- (+ extended-right
                          (* track (+ space width)))
                       width)
                    (+ (car t-in) (/ width 2)))
   (rect 'NP (- extended-right power 3 power 3 (/ width 2))
             (- (car t-in) 1)
             (- (+ extended-right (* track (+ space width)))
                    (/ width 2))
             (+ (car t-in) 1)))))
```

APPENDIX C

SOURCE CODES FOR TEST CIRCUITS

A. MEMORY

```
(program memory 2
  (def 11 power)
  (def 1 ground)
  (def 2 phia)
  (def 3 phib)
  (def 4 phic)
  (def on signal input 5)
  (def reset signal input 6)
  (def a port input (7 8))
  (def b register)
  (def c port output (9 10))
  (proces proc1 0
   off
     (cond (on (go start))
           (t (go off)))
   start
     (cond (on (setq b a)
               (setq c b)
               (go start))
           (t (setq b a) (go off)))))
```

B. TEST

```
(program test 2
   (def 10 power)
   (def 1 ground)
   (def 2 phia)
   (def 3 phib)
   (def 4 phic)
   (def on signal input 5)
   (def reset signal input 6)
   (def a port input (7 8))
   (def b register)
   (def c port output (9 10))
   (process proc1 0
     run (setq b a)
         (setq c b)
         (go run)))
```

C. MULTIP4

```
(program multip4 4
  (def 1 ground)
  (def ain port input (2 3 4 5))
  (def a0 register)
  (def a1 register)
  (def a2 register)
  (def bin port input (6 7 8 9))
  (def b0 register)
  (def b1 register)
  (def b2 register)
  (def res port output (10 11 12 13))
  (def r0 register)
  (def r1 register)
  (def r2 register)
  (def 14 phia)
  (def 15 phib)
  (def 16 phic)
  (def reset signal input 17)
  (def 18 power)
  (always
   (cond ((bit 0 bin) (setq r0 (>> (bit 0 ain) ain)))
         (t (setq r0 0)))
   (cond ((bit 1 bo) (setq r1 (>> (bit 0 (+ r0 a0)) (+ r0 a0))))
         (t (setq  r1 (>> (bit 0 r0) r0))))
   (cond ((bit 2 b1) (setq r2 (>> (bit 0 (+ r1 a1)) (+ r1 a1))))
         (t (setq r2 (>> (bit 0 r1) r1))))
   (cond ((bit 3 b2) (setq res (>> (bit 0 (+ r2  a2))
                                   (+ r2 a2))))
         (t (setq res (>> (bit 0 r2) r2))))
   (cond (reset (setq a0 0)
                (setq b0 0)
                (setq a1 0)
                (setq b1 0)
                (setq a2 0)
                (set qb2 0)
         (t     (setq a0 ain)
                (setq b0 bin)
                (setq a1 a0)
                (setq b1 b0)
                (setq a2 a1)
                (setq b2 b1)))))
```

D. TAXI

```
(program taxi 8
 (def 17 power)
 (def 1 ground)
 (def 2 phia)
 (def 3 phib)
 (def 4 phic)
 (def timer register)
 (def fare register)
 (def reset signal input 5)
 (def time-on signal input 6)
 (def hire signal input 7)
 (def mile-mark signal input 8)
 (def display port inp[ut (9 10 11 12 13 14 15 16))
 (def charge-time signal internal)
 (def maximum-time constant 100)
 (def base-fare constant 20)
 (def cost-per-mile constant 50)
 (def cost-per-time constant 10)
 (process time-clock 0
  off
   (cond (time-on (setq timer 0) (go on))
         (t (go off)))
  on
   (cond (time-on (cond ((= timer maximum-time)
                         (setq timer 0)
                         (signal charge-time)
                        (t (setq timer (1+ timer))))
                  (go on))
         (t (setq timer 0) (go off))))
(process fare-clock 0
 for-hire
  (cond (hire (setq fare base-fare) (go hired))
        (t (go for-hire)))
 hired
  (par (cond ((not hire) (go for-hire))
             ((and charge-time mile-mark)
              (setq fare (+ (+ fare cost-per-mile)
                            cost-per-time))
              (go hired))
             (charge-time
              (setq fare (+ fare cost-per-time))
              (go hired))
             (mile-mark
              (setq fare (+ fare cost-per-mile))
              (go hired))
             (t (go hired)))
       (setq display fare))))
```

LIST OF REFERENCES

1. Weste, Neil H. E., and Eshraghian, Kamran., *Principles of CMOS VLSI Design, A Systems Perspective.* Reading, MA: Addison-Wesley Publishing Co., 1985.

2. Siskind, J. M., Southard, J. R., and Crouch, K. W., *Generating Custom High Performance VLSI Designs From Succint Algorithmic Descriptions.* Proceedings, Conference on Advanced Research in VLSI, January 1982.

3. Carlson, D. J., *Applications of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers.* M.S. Thesis, Naval Postgraduate School, Monterey, CA, June 1984.

4. Froede, A. O., *Silicon Compiler Design of Combinational and Pipeline Adder Integrated Circuits.* M. S. Thesis, Naval Postgraduate School, Monterey, CA, June 1985.

5. Larabee, R. C., *VLSI Design With the MacPitts Silicon Compiler.* M. S. Thesis, Naval Postgraduate School, Monterey, CA, September 1985.

6. Malagon-Fajar, M. A., *Silicon Compilation Using a LISP-Based Layout Language.* M. S., Thesis, Naval Postgraduate School, Monterey, CA, June 1986.

7. Weist, E. L., *A Flowcharting System and Compiler Interface for MacPitts.* M. S. Thesis, Naval Postgraduate School, Monterey, CA, June 1986.

8. Mullarky, A. J., *CMOS cell Library for a Silicon Compiler.* M. S. Thesis. Naval Postgraduate School, Monterey, CA, March 1987.

9. Malagon, E. G., *Techology Upgrade of a Silicon Compiler.* M. S. Thesis, Naval postgraduate School, Monterey, CA, June 1987.

10. Baumstarck, J. E., *SCMOS Silicon Compiler Organelle Design and Insertion.* M. S. Thesis, Naval Postgraduate School, Monterey, CA, December 1987.

11. Mead, C. A., and Conway, L. A., *Introduction to VLSI Systems.* Addison-Wesley Publishing Co., Reading, MA, 1980.

12. Computer Science Division, EECS Department University of California at Berkeley, *1986 VLSI Tools; Still More Works by the Original Artists*, Report No. UCB/CSD 86/272, December 1985.

13. Kelly, M. F., *Comparative Router Performance.* Ph. D. Thesis, University of California, Livermore, CA, 1977.

14. Lee, C., *An Algorithmic for Path Connections and Its Applications*, IRE Transactions on Electronic Computers (September 1961), pp. 346-365.

15. Hashimoto, A., and Stevens, J., *Wire Routing by Optimizing Channel Assignment Within Large Apertures*, Proceedings Design Automation Workshop (1971), pp. 165-169.

16. Rivest, R. L., and Fiduccia, C. M., *A "Greedy" Channel Router*, 19th DesignAutomation Conference (1982), pp. 418-424.

17. Deutsch, D. N., *A "Dogleg" Channel Router*, Proceedings 13th Design Automation Conference (1976), pp. 425- 433.

18. McGehee, R. K., *A Practical Moat Router*, 24th Design Automation Conference (1987), pp. 216-221.

19. Harmon, J. E., *Automated Design of a Microprogrammed Controller for a Finite State Machine*, M. S. Thesis, Naval Postgraduate School, Monterey, CA, Work in progress.

20. Wyatt, J. L. Jr., *The Practical Engineer's No-nonsense Guide to On-Chip Signal Delay Calculations*, VLSI Memo No. 87- 381, Massachusetts Institute of Technology, May 1987.

BIBLIOGRAPHY

1. Carré, Bernard, *Graphs and Networks*, Clarendon Press, Oxford, 1979.

2. Crouch, K. W., *L5 User's Guide*, Massachusetts Institute of Technology Lincoln Laboratories Project Report RVLSI-5, 7 March 1984.

3. Engineering Staff of American Micro-Systems, Inc., *MOS Integrated Circuits*, Van Nostrand Reinhold Co., 1972.

4. Joobbani, Rostam., *An Artificial Intelligence Approach to VLSI Routing*, Kluwer Academic Publishers, 1986.

5. Mukherjee, Amar, *nMOS & CMOS VLSI Systems Design*, Prentice-Hall, Englewood Cliffs, N.J., 1986.

6. Wilensky, Robert, *LISPcraft*, New York, NY: W. W. Norton & Co., 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	1
5.	Dr. D. E. Kirk, Code 62KI Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	2
6.	Dr. H. H. Loomis, Jr., Code 62LM Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	2
7.	Dr. D. C. Yang, Code 62YA Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	2
8.	Dr. M. Zyda, Code 52MZ Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	1

		No. Copies
9.	Mr. P. Blankenship Massachusetts Institute of Technology Lincoln Laboratory P. O. Box 73 Lexington, MA 02173-0073	1
10.	Mr. J. O'Leary Massachusetts Institute of Technology Lincoln Laboratory P. O. Box 73 Lexington, MA 02173-0073	1
11.	Dr. T. Bestul Naval Research Laboratories Code 7590 Washington, D.C. 20375	1
12.	Mr. A. DeGroot Lawrence Livermore National Laboratory P.O. Box 808 Livermore, CA 94550	1
13.	Dr. A. Ross Naval Research Laboratory, Code 9110 4555 Overlook Ave. SW Washington, D.C. 20375	1
14.	CDR David Southworth Office of Naval Technology, Code ONT227 800 N. Quincy (BT #1) Arlington, VA 22217-5000	1
15.	Mr. James Hall Office of Naval Technology, Code ONT20P4 800 N. Quincy (BT #1) Arlington, VA 22217-5000	1
16.	LT. D. Carleton, USN SMC #1493 Naval Postgraduate School Monterey, CA 93943	1

Thesis
R3662
c.1
Rexach
A pad router for the
Monterey Silicon Compiler.

Thesis
R3662
c.1
Rexach
A pad router for the
Monterey Silicon Compiler.

DEMCO

thesR3662
C.1
A pad router for the Monterey Silicon Co
3 2768 000 78908 5
DUDLEY KNOX LIBRARY